Supporting Policy-Based Contextual Reconfiguration and Adaptation in Ubiquitous Computing

Lachhman Das Dhomeja
l.d.dhomeja@sussex.ac.uk

November, 2010

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy (DPhil) in the School of Informatics, University of Sussex, Brighton, UK

Abstract

In order for pervasive computing systems to be able to perform tasks which support us in everyday life without requiring attention from the users of the environment, they need to adapt themselves in response to context. This makes context-awareness in general, and context-aware adaptation in particular, an essential requirement for pervasive computing systems. Two of the features of context-awareness are: contextual reconfiguration and contextual adaptation in which applications adapt their behaviour in response to context. We combine both these features of context-awareness to provide a broad scope of adaptation and put forward a system, called Policy-Based Contextual Reconfiguration and Adaptation (PCRA) that provides runtime support for both.

The combination of both context-aware reconfiguration and context-aware adaptation provides a broad scope of adaptation and hence allows the development of diverse adaptive context-aware applications. However, another important issue is the choice of an effective means for developing, modifying and extending such applications. The main argument forming the basis of this thesis is that we advocate the use of a policy-based programming model and argue that it provides more effective means for developing, modifying and extending such applications.

This thesis addresses other important surrounding issues which are associated with adaptive context-aware applications. These include the management of invalid bindings and the provision of seamless caching support for remote services involved in bindings for improved performance. The bindings may become invalid due to failure conditions that can arise due to network problems or migration of software components, causing bindings between the application component and remote service to become invalid. We have integrated reconfiguration support to manage bindings, and seamless caching support for remote services in PCRA.

This thesis also describes the design and implementation of PCRA, which enables development of adaptive context-aware applications using policy specifications. Within PCRA, adaptive context-aware applications are modelled by specifying binding policies and adaptation policies. The use of policies within PCRA simplifies the development task because policies are expressed at a high-level of abstraction, and are expressed independently of each other. PCRA also allows the dynamic modification of applications since policies are independent units of execution and can be dynamically loaded and removed from the system. This is a powerful and useful capability as applications may evolve over time, i.e. the user needs and preferences may change, but re-starting is undesirable. We evaluate PCRA by comparing its features to other systems in the literature, and by performance measures.

Acknowledgements

First of all, I would like to thank my supervisors, Dr. Dan Chalmers and Dr. Ian Wakeman for their valuable guidance over the past four years. I am particularly grateful to Dan Chalmers, whose constructive criticism and countless meetings with him helped me both to structure my thoughts and to keep myself focused on such an interesting research topic, which eventually led to this thesis.

Many thanks to other members of the Foundations of Software System Group, particularly Dr. Des Watson, Yasir Arfat Malkani, Roya Feizy, Stephen Naicken, Aeshah Alsiyami, Renan Krishna, Ben Horsfall and Dr. Eskindir Asmare. Special thanks to Simon Fleming, Danny Matthews and James Stanier from this group for proof-reading this thesis. I would also like to extend my thanks to my past colleagues, Dr. Jon Robinson, and Dr. Jian Li, and also to Dr. Kevin Twiddle, senior research fellow at Imperial College, London for answering my emails about Ponder2.

I am greatly indebted to my parents for being so supportive and loving throughout my life. My sincere thanks are due to my wife for being very understanding, caring, loving and helpful. Thanks to my children – Maanta Dhomeja and Ronat Dhomeja for being source of entertainment and recreation during difficult times of my research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In 1991, Mark Weiser outlined his vision of ubiquitous computing [85], now also known as pervasive computing, in which he predicted that computing would no longer be confined to desktops, but become ubiquitous and invisible to the user. Discussing Weiser's vision, Satyanarayanan [86] defined invisibility as the *"complete disappearance of pervasive computing technology from a user's consciousness"* and approximated this to "*minimal user distractions"*. In order for pervasive computing systems to be able to perform tasks which support us in everyday life with minimal or no user distraction, they need to adapt themselves in response to context. This makes context-awareness in general and context-aware adaptation in particular, an essential requirement for pervasive computing systems.

There have been a number of definitions of context and context-aware computing (context-awareness) proposed within the literature (we discuss in chapter 2). Two of the categories of context-awareness identified and discussed by the researchers are contextual reconfiguration and contextual adaptation. These categories identify that an application can modify (adapt) its behaviour in response to context and thus any applications belonging to these categories can be considered to be adaptive context-aware applications. The other context-aware applications that belong to the other two categories (contextual sensing and contextual augmentation in existing terminology) cannot be considered to be adaptive context-aware applications because these applications don't adapt themselves in response to context; instead either contextual information describing the current context (e.g., temperature, location, etc.) is presented to a user, or context is associated with data (e.g. records of objects surveyed can be associated with location, meeting notes can be associated with people in the meeting, etc.). Based on the above discussion we argue that adaptive context-aware applications are a subset of context-aware applications. Efstratiou [2] has also made similar arguments while defining adaptive context-aware applications.

Contextual reconfiguration is the process of restructuring or reconfiguring the software components of the applications to realize new behaviour which may be required to fulfil user needs or to enrich the user experience. This can be achieved by discovering service(s) based on context and binding them to application components. For example, a user may want to have her messages displayed or printed to the nearest rendering device to her location. This requires discovering a device based on the location of the user and binding to it, and then sending messages to the bound device. As another example, contextual reconfiguration may be used to enrich the experience of a mobile user by providing her with a service of interest with respect to her changed location without requiring any cooperation from her. For instance, when the user is standing near a cinema, a movie information service could send information about the movies being exhibited in that cinema.

Unlike contextual reconfiguration where adaptation is achieved by reconfiguring application code (e.g. by discovering service(s) based on context and binding them to application component), contextual adaptation is the process of modifying the application behaviour through the modification of the behaviour of the component/service involved in the binding, in response to context. Chalmers [5] has separated two cases in contextual adaptation: (1) when context is involved in triggering an action and (2) when context is involved in modifying the actions, which have been caused separately. As a result of this distinction, he has divided contextual adaptation into two forms: context-triggered actions [4] and contextual mediation—using context to modify a service. In the context of Chalmers's work, adaptation which involves modifying a service based on context is contextual mediation where the best variant of data elements is selected based on context of use. Using context to modify a service may involve other processes than contextual mediation. For example, in a simple home lighting scenario the light service may be modified to adjust the light value to some user-preferred value based on the user's activity. Regardless of what kind of adaptation the modification of actions serve, an important point to note is that triggering actions based on context (context-triggered actions) and modifying these actions based on context (e.g. contextual mediation) are two separate tasks, thus two separate cases. Our work on contextual adaptation is inspired by Chalmers's philosophical view on contextual adaptation and thus includes both context triggered actions and modifications of these actions.

The field of context-aware adaptation has been widely studied and applied in mobile environments to address inherent limitations of mobile technology (such as varying network quality, limited battery life), where the service is adapted in response to context (such as resource variation, small screen size, limited battery power, etc.) to retain the usefulness of mobile applications without any intervention from users. While there has been much work on

contextual adaptation in mobile environments where mobile applications adapt to resource variability, we focus on a set of applications in other pervasive computing environments (e.g. domestic environment), where contextual adaptation is required to be performed in response to other context triggers, such as environmental context (light level, noise level, temperature level, etc.) and user context (user presence, user activity, etc.).

We believe that both the contextual reconfiguration and contextual adaptation features of context-awareness are interesting, as applications in these categories adapt their behaviour according to context, thus helping to minimise user distractions. Both of these adaptive features of context-awareness provide different kinds of adaptations and can individually satisfy the adaptation requirements of various adaptive context-aware applications. However, we envision many adaptive context-aware scenarios (see chapter 4 for various adaptive context-aware applications that we have implemented) whose adaptation requirements cannot be satisfied by either of these features alone. In order to provide a broad scope of adaptation, we combine both features of context-awareness—contextual reconfiguration and contextual adaptation and present a system, called Policy-Based Contextual Reconfiguration and Adaptation (PCRA), which provides runtime support for both. As discussed before, we consider two separate cases in contextual adaptation: context-triggered actions and modification of these actions in response to context. Contextual adaptation support integrated within PCRA employs a parameter adaptation approach for modifying triggered actions in response to context, where the service behaviour is modified through parameter adjustments. In chapter 2, we have identified and discussed various adaptation approaches used in the literature to achieve dynamic adaptation, and the service parameter adaptation is one of these.

An important issue associated with adaptive context-aware applications is that of the invalidity of bindings. There come times when bindings, created by reconfiguration support, become invalid for reasons including sudden non-availability of the bound service (due to power failure at the hosting device where the bound service is running), or the bound service has been moved to some other location over the network for load-balancing purposes, or it has been moved closer to the entity accessing the bound service, in order to save bandwidth. In all these situations, the real proxy/stub of the bound service becomes invalid, causing all the bindings to this service to become invalid. Adaptive context-aware applications are complex to develop, maintain and modify, and the issue of managing these binding failures further complicates development efforts. In order to reduce the complexity involved in developing, maintaining and modifying adaptive context-aware applications, we propose to relieve the application developer from the responsibility for managing bindings and to delegate this to our system, PCRA.

One of the steps in the reconfiguration process involves a remote lookup to discover the service(s) based on context. This remote lookup process provides the largest contribution to reconfiguration time. This is due to fact that the remote calls are much slower than local calls. In adaptive context-aware applications, this may turn out to be undesirable in terms of user experience. Moreover, other distributed applications using the network may be affected as every remote method call decreases the amount of bandwidth available on the network. To address this problem we propose and implement seamless caching support of virtual stubs (we discuss virtual stubs in chapter 3) within PCRA for improved performance.

The provision of a broader scope of adaptation satisfies different adaptation requirements of the applications and hence allows the development of diverse adaptive context-aware applications. However, another core issue is the simplification of development and dynamic modification of applications. Many research efforts [21,22,24,25,69,77] focus on this and provide means to simplify development. The approaches offered by these systems are in the form of Application Programming Interface (API) or the development of specially designed languages, which include high-level constructs to code adaptive context-aware applications. Our evaluation results (chapter 6) and review of various related systems (chapter 7) suggest that the specifically designed languages contribute better towards the goal of simplifying development, though APIs provide a more flexible model but at the cost of ease of use. While approaches based on specifically designed languages simplify the development task due to the use of high-level adaptation directives, these approaches have one limitation in that adaptation directives are hard-coded inside the applications. This limitation introduces inflexibility in the sense that it does not allow dynamic modification. To modify the application, it needs to be stopped, modified and then restarted. To address this limitation, we advocate and present a policy-based programming approach for developing, modifying and extending such applications, and use a specialized declarative policy system, Ponder2 [26,80,94], for specifying binding and adaptation policies. In the evaluation chapter, we show that the policy-based programming approach provides a more effective means for developing adaptive context-aware applications than an API-based approach or specifically designed languages.

1.2 Contributions

In this thesis, we present the Policy-based Contextual Reconfiguration and Adaptation (PCRA) system that enables rapid development and maintenance of adaptive context-aware applications and also allows dynamic modification of the applications. To provide a broader

scope of adaptation and to simplify the task of developing, maintaining and modifying adaptive context-aware applications, PCRA provides several features:

- **Scope of adaptation:** In order to provide a broader scope of adaptation, PCRA supports following types of adaptation.
    - **Context-aware reconfiguration of bindings**: PCRA system provides runtime support for context-aware reconfiguration of bindings. This runtime support responds to binding policies and performs reconfiguration activities. These activities involve discovering service(s) based on context and then creating bindings between the user and discovered service(s).

    - **Context-aware adaptation of service:** PCRA provides runtime support for context-aware adaptation of service, which involves both triggering the behavior of the bound service in response to context (i.e., context-triggered actions) and modifying the triggered behavior of that service in response to context. PCRA uses parametric adaptation for realizing the latter. The policies are written to encode adaptation behavior, and service adaptation runtime support respond to these policies and realize adaptation.

    - **Reconfiguration to manage invalid bindings:** The bindings created by reconfiguration of bindings runtime support can become invalid for various reasons. PCRA also provides reconfiguration support to manage invalid bindings. This reconfiguration support relieves an application developer from the responsibility for managing invalid bindings, thus contributing towards simplifying the task of developing, maintaining and modifying the adaptive context-aware applications.

- **Support for caching of virtual stubs for improved performance:** PCRA provides runtime support for caching virtual stubs for improved performance, where, as a part of the reconfiguration process, the remote service is discovered, a virtual instance created and initialized with the discovered service and then cached locally. When the application needs to create the binding to this service again, the corresponding virtual stub is obtained from the local cache directly without the need for a remote call. This significantly reduces reconfiguration time, hence improves the system performance.

- **Use of the policy-based programming approach:** We advocate and propose a policy-based programming approach to reconfiguration and adaptation. We argue that the policy-based approach is an efficient way of developing, modifying and extending adaptive context-aware applications as opposed to specifically designed languages, which provide high-level languages constructs for adaptation and an API-based approach, where adaptation support is provided through an API. The effectiveness of the policy-based approach comes from: (1) policies are expressed declaratively at a high-level of abstraction (i.e., application developers are not required to know low level details of reconfiguration / adaptation mechanisms) and independently of each other, thus simplifying development and (2) policies allow dynamic modification of adaptive context-aware applications as policies are independent units of execution and can be dynamically loaded and unloaded from the system. This is a powerful and useful feature of the policy-based approach as applications may evolve over time (i.e., the user needs and preferences may change) but re-starting is undesirable, unlike both specifically designed languages and API-based approaches that do not support dynamic modification of adaptive context-aware applications.

1.3 Thesis Outline

Chapter 2 presents the background to this work. We review the literature relating to two parts of our research problem—context-awareness and policies. We also identify from the literature various adaptation approaches used to realize dynamic adaptation. From the literature review, we set the scope of research work and establish its place in the field.

Chapter 3 provides an introduction to PCRA, where we provide its high-level architecture and briefly describe various elements of it. We provide a description of each of our contributions supported by PCRA and provide an architectural overview of each, and then discuss how each contribution is realized through the architectural components involved.

Chapter 4 provides a high-level description of various hypothetical example scenarios that we have designed and implemented. These scenarios are used to examine our contributions as well as a basis for the evaluation of our approach to the development of adaptive context-aware applications and its comparison with Scooby and One.World.

Chapter 5 describes the prototype implementation of our reconfiguration and adaptation infrastructure integrated within PCRA. We describe the design and implementation of our GUI-based system utility. This chapter also provides a description and implementation of various simulated context monitors that are used in the implementation of example scenarios.

Chapter 6 provides the evaluation of PCRA. In this chapter we evaluate our policy-based programming approach and compare it to a specifically designed language for service composition/reconfiguration (Scooby) and an API-based approach (One.World). We also study the performance of two main features of PCRA: contextual reconfiguration and contextual adaptation.

Chapter 7 provides an overview of related systems. We describe various core concepts involved in each system, and which adaptation approaches have been used by these systems to realize dynamic adaptation. This chapter also discusses which approach is provided by these systems to develop adaptive context-aware applications, and then briefly compare it with our approach. We also review some of the approaches used in the systems to handle the issue of updating invalid references and compare with them our approach to handling this issue.

Chapter 8 summarizes the contributions of this research work and explores directions for future work.

Chapter 2

Background

Context awareness, in general and context-aware adaptation in particular, is a central aspect of mobile and ubiquitous computing applications, characterizing their ability to adapt and perform tasks based on context. Context-aware computing was pioneered by researchers at Xerox PARC Laboratory [37-39] under the vision of ubiquitous computing [40], also known as pervasive computing. Since then, this area has been widely studied and many context-aware systems have been built to demonstrate the usefulness of this research area. A summary of contributions made in this area can be found in [3,9]. Context-aware computing is a computing paradigm in which applications can discover and take advantage of contextual information (such as user presence, user activity, user location, device characteristics, resource variation, time, light and noise levels, etc.). For example, in mobile computing it is used to help deal with the technological limitations (such as low CPU power, small display, low battery, and low communication bandwidth) so that the usefulness of an application is retained under various circumstances. Due to the availability of new environment monitoring technologies (location tracking, service discovery etc.), it is possible to enrich the experience of a mobile user by providing her with information of interest based on her current location. For instance, when the user is standing near a cinema, a movie information service should send information about the movies being exhibited in that cinema etc.

The smart home is a place which adapts in order to improve comfort, safety, security and resource use to its occupants. In smart environments such as smart homes, it is the context-awareness that makes a home smart. This is achieved by interconnecting sensors, computing, appliance and services. While some benefits may come from a programmed home, "smartness" arises from context-awareness where a reprogrammable home adapts to context (occupancy, activities, weather, etc.).

Recently context-awareness has been applied in body sensor networks [41,42] to provide continuous monitoring of medical conditions of a patient such as heart-rate and glucose level, and to provide alerts to healthcare personnel or the patient herself to suggest a course of action in case of abnormal conditions.

This chapter is organised as follows. Section 2.1 and 2.2 give an overview of definition of context and context-aware computing given by various researchers. Section 2.3 discusses

adaptive context-aware applications. Section 2.4 gives an overview of various adaptation approaches used in the literature to realize dynamic adaptation. Section 2.5 presents and describes our reconfiguration approach to managing bindings. Section 2.6 provides an overview of policies and context-awareness. Finally, section 2.7 gives a summary of this chapter.

2.1 A Definition of Context

There has been much debate on the definition of context and context-aware computing in the research community and that has given rise to a variety of definitions of context and consequently context-aware computing.

Schilit et al. [4] define context by enumerating examples of context and claim that important aspects of context are: where are you, who you are with and what resources are nearby. They define context to be constantly changing execution environment such as communication bandwidth, processors available for a task, etc. They divide context into three categories:

- *Computing context*, such as communication bandwidth, network connectivity, available processors, communication costs, and nearby resources such as printers, and displays.
- *User context*, such as the user location, the user's profile, people nearby.
- *Physical context*, such as lighting, noise levels, traffic conditions, and temperature.

Brown et al. [43] also define context by examples and describe it as location, the time of the day, temperature, etc.

Chen and Kotz [9] argue that context in mobile computing has two different aspects. One aspect is about the characteristics of surrounding environment that determine the behaviour of an application, while the other aspect of context is relevant to an application but not critical, where it is not necessary for the application to adapt its behaviour to the context except to display them to interested users. Based on this observation they define context as *"Context is the set of environmental states and settings that either determines an application's behaviour or in which an application event occurs and is interesting to the user"*. Furthermore, they divide context into two categories based on these two different aspects of context: active context, which directly influences the application behaviour; and passive context that is relevant but not critical to the application.

Based on the definition of context in the oxford dictionary (1995), which defines context to be *"the circumstances relevant to something under consideration"*, Chalmers [5] takes computing oriented view of this definition and considers *"something under*

consideration" to be some interaction with a computing device, and then define context to be "*Context is the circumstances relevant to the interaction between a user and their computing environment*".

Dey and Abowd [3] define context to be "*Any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the applications themselves*".

It can be observed from the above discussion that out of all the definitions of context given by various researchers, the definition of context given by Chalmers [5] and Dey and Abowd [3] is broader in the sense that it can include any information that characterizes the situation of a participant in an interaction, be that resource variation, mobility aspects, user location, preferences of the user, user activity, lighting, noise level, temperature, etc.

2.2 Uses of Context

Context-aware computing was first discussed by Schilit and Theimer [39], where they define context-aware computing to be software that "adapts according to its location of use, the collection of nearby people, as well as changes to those objects over time". Since then, there have been a number of definitions of context-aware computing in the literature by various researchers, such as Pascoe [8], Chen and Kotz [9], Chalmers [5] and Dey and Abowd [3]. In order to help identify and define core features of context-awareness, these researchers have categorized its features.

Schilit et al. [4] divided context-aware application into four categories: *Proximate Selection, Automatic Contextual Reconfiguration, Contextual Commands and Context-triggered Actions.*

Pascoe [8] proposed a taxonomy of context-aware features as follows: *Context Sensing, Contextual Adaptation, Contextual Resource Discovery and Contextual Augmentation*.

Dey and Abowd [3] combines the ideas from both these taxonomies and takes into account some differences, and proposes categorization as follows:

(1) presentation of information and services to a user;

(2) automatic execution of a service; and

(3) tagging of context to information for later retrieval

The first category is the combination of Schilit's proximate selection and contextual commands, and it also includes Pascoe's contextual sensing. The second category is the same as

Pascoe's contextual adaptation and Schilit's context-triggered actions. And the third category is the same as Pascoe's contextual augmentation. The important point to note is that Dey and Abowd's proposed categorization does not explicitly have a separate category that could map Schilit's contextual reconfiguration and Pascoe's contextual resource discovery. They consider resource exploitation or resource discovery no different from choosing a service based on context. They argue that resource exploitation or resource discovery can be the presentation of resource information (e.g., list of printers) to the user based on context or automatic execution of a service (e.g., printer service) for the user based on context. In the former it falls into the first category and in the latter it falls into the second category. This indicates that the second category, in addition to being the same as Schilit's context-triggered actions and Pascoe's contextual adaptation, implicitly includes Schilit's contextual reconfiguration and Pascoe's contextual resource discovery.

Chalmers identifies five uses of contextual information in pervasive computing environments, drawing from Dey and Abowd and Schilit et al.: contextual sensing, contextual augmentation, contextual resource discovery, context-triggered actions and contextual mediation.

Chalmers [5] has separated two cases in contextual adaptation: 1) when context is involved in triggering an action and 2) when context is involved in modifying the actions, which have been caused separately. As a result of this distinction, Chalmers has divided contextual adaptation into two forms: context-triggered actions [4] and contextual mediation—using context to modify a service. In the context of Chalmers's work, adaptation which involves modifying a service based on context is contextual mediation where the best variant of data elements is selected based on context of use. Using context to modify a service may involve other than contextual mediation. For example, in a simple home lighting scenario the light service may be modified to adjust the light value to some user preferred value based on the user's activity; or in some other context-aware scenario the TV volume may be reduced when someone is talking on the phone. Irrespective of what kind of adaptation the modification of actions serve, an important point to note is that triggering actions based on context (context-triggered actions) and modifying these actions based on context(e.g., contextual mediation) are two separate tasks thus two separate cases.

Following the above discussion, we argue that when context is involved in triggering an action it is acting as an input to a context-aware system and when it is involved in modifying these actions it is acting as a modifier to the context-aware system. The difference between context-triggered actions and modification of these actions can be understood through this simple example. When the user enters the room, the light is turned ON (context-triggered

action—the light is turned ON in response to the user presence context) and then the light is adjusted to some value based on the user's activity (modification of action----the light value is adjusted to a certain value in response to activity context). In this example user presence context is an input since it is involved here to trigger the action (turning the light ON), while activity context is a modifier since it is involved in modifying the action (adjusting the light value). Our work on contextual adaptation is inspired by Chalmers's philosophical view on contextual adaptation and thus includes both context triggered actions and modifications of these actions.

Many researchers have attempted to provide the definition of context and context-aware computing. A summary definition of context-aware computing is given by Dey and Abowd, which states:

> *"A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task."*

2.3 Adaptive Context-Aware Applications

Dey and Abowd's second category of context-awareness identifies that an application can modify (adapt) its behaviour in response to context and thus any application in this category can be considered to be an adaptive context-aware application. As discussed earlier, contextual reconfiguration/contextual resource discovery has been explicitly defined by other researchers as a separate context-aware feature, while Dey and Abowd argue that reconfiguration/discovery is no different from automatic execution of a service, as discussed earlier. We would like to explicitly mention that applications belonging to this category can be considered to be adaptive context-aware applications in the sense that application behaviour is modified by discovering a service based on context and then binding with it to realize new behaviour based on context. Context-aware applications that belong to the other two categories (contextual sensing and contextual augmentation in existing terminology) cannot be considered to be adaptive context-aware applications because applications don't adapt themselves in response to context, but either contextual information describing the current context(e.g., temperature, location, etc.) is presented to a user or context is associated with data(e.g. records of objects surveyed can be associated with location, meeting notes can be associated with people in the meeting, etc.). The above discussion reveals that adaptive context-aware applications are a subset of context-aware applications, but not vice versa. Efstratiou [2] has also made a distinction between adaptive context-aware applications and context-aware applications and has made similar arguments.

The field of context-awareness is broad and there has been much work on all categories of context-awareness. Contextual adaptation in mobile environments is used to handle inherent

restrictions of mobile technology and other issues such as heterogeneity of clients/devices, where the service is modified in response to context (such as resource variation, limited battery power, small screen size, etc.) to retain the usefulness of applications without any intervention from the users of the applications [1,2,10-17,44,45,96-98]. While early research focused on contextual adaptation where applications were required to adapt to resource variability, there are other pervasive computing environments (e.g., domestic environment) where applications are required to adapt to other contextual triggers such as environmental context(light level, noise level, temperature level) and user context (user presence, user activity, etc.). We focus on a set of applications scenarios where contextual adaptation is required to be carried out in response to environmental context and user context.

In contrast to contextual adaptation, contextual reconfiguration involves restructuring or reconfiguring the structural parts of the application, in order to achieve new behaviour of the application required in the current operating environment. The field of dynamic reconfiguration is broad and has been widely studied and applied in research domains such as stationary distributed systems [46-49] and mobile and pervasive applications [1, 23,50-52]. In stationary distributed systems this has been applied to achieve load-balancing, fault-tolerance, to improve performance or to harden security in response to attack, without taking the system offline. Reconfiguration is an important approach to apply in distributed systems where the high availability of the system is an important requirement. In mobile and pervasive systems contextual reconfiguration may be used to enrich the experience of a mobile user by providing her with a service of her interest with respect to her changed location without requiring any cooperation from her.

We believe that contextual reconfiguration and contextual adaptation features of context-awareness are more interesting as applications in these categories adapt their behaviour according to context, thus helping to eliminate unnecessary user cooperation, making technology as “calm” as possible. While there has been much work on both contextual reconfiguration and contextual adaptation individually, we envision many adaptive context-aware applications that involve both these features of context-awareness and focus on combining both context-aware features. We put forward a system that provides runtime support for both contextual reconfiguration and contextual adaptation.

In the next section, we review various adaptation approaches that have been used in the field of mobile and ubiquitous computing to realize dynamic adaptation.

2.4 Adaptation Approaches

After a review on the literature [6,28,53,54,92], we have found the following adaptation techniques that have been studied and applied in various research domains. Each of these adaptation approaches enables powerful adaptations.

- Dynamic Association and Disassociation of Non-functional/Low-level Services
- Component Reconfiguration
- Dynamic Reconfiguration of Application Components
- Dynamic Reconfiguration of Remote Method Invocation
- Code Mobility

2.4.1 Dynamic Association and Disassociation of Non-functional Concerns/Low-level Service

This is the one of the adaptation approaches used for dynamic adaptation in response to a change in context. This approach is based on the separation of concerns principle [36,93] where non-functional concerns, also called crosscutting concerns, of an application are implemented separately from core application concerns. The core behaviour of applications is sometimes referred to as business logic. Both the non-functional concerns and functional concerns, and low-level services exist independently of each other. This separation facilitates development, maintenance and also software components reuse. The examples of non-functional concern are access control, persistence management etc., and examples of low-level services are caching service, encryption service and data compression services etc. Traditionally, these services are offered by the underlying middleware and used by the applications.

The core idea behind this approach is the separation between what depends on changing context, which may be non-functional concern or a low-level service of the underlying middleware and what does not, which is the core functional concern. The application is then adapted by dynamically associating the non-functional/low-level service with the core functional concern in response to changing context. An example of dynamically associating the non-functional concern with the core application component would be associating persistence with a mobile client in response to the battery of the device running low so that data is preserved in case of power failure. An example of associating a low-level service with the core application component would be associating the caching service with the mobile client when bandwidth runs low so that fields of remote service could be used locally, in order to save the limited bandwidth.

A field which is closely related to the dynamic adaptation achieved through this adaptation approach is Aspect Oriented Programming Approach) (AOP) [55,56]. Hence, these approaches are commonly referred as AOP approaches. In these approaches, the non-functional concern is implemented as an aspect and the application is adapted by weaving and unweaving aspects into and from core functional concerns. However, there are other systems [12,13,18], which exploit this adaptation approach, using reflection and meta-programming techniques [57] to enable adaptations.

2.4.2 Component Reconfiguration

This is another powerful adaptation technique, which is also referred to as service parameter adaptation in the literature [6,92]. In this approach, the dynamic adaptation is achieved by tuning the behaviour of the service through an adjustment of its parameter(s) in response to a change of context. The classical example, which is most commonly used in the literature, is of the TCP protocol and the way it modifies its behaviour by changing values that control the congestion window management and retransmission in response to network conditions. Other examples of this approach would be changing the light level of a room in response to user activity, e.g., sleeping, reading etc.; changing the size of the cache and reducing the frame-rate or resolution of an image in response to decreasing bandwidth. This adaptation technique has been applied in many systems to achieve dynamic adaptation in response to context, for instance [12,14-16,22,52].

2.4.3 Dynamic Reconfiguration of Application Code

Unlike other adaptation approaches as discussed above, where adaptation is achieved by adapting the functional concerns of an application, low-level service or a service/component through parameters, this approach involves adapting application code itself by adding, removing and replacing core application components. Several approaches for reconfiguration of an application have been proposed that target both stationary distributed systems and mobile and pervasive systems, and we have mentioned some systems in section 2.3 that use this technique. Depending on the type of applications targeted, application reconfiguration can be carried out in the following two ways: (1) Application configurations are already specified through some means, e.g., through markup languages (such as XML [89], ADML [87]), general purpose modelling notations (such as UML [88]) and Architectural Description Languages (such as Acme [91], Darwin [90]), which are suited to particular contextual information. In response to context, a configuration is selected and implemented through reconfiguration actions (addition, removal or replacement of components). The core idea behind this is to make runtime change in

response to context through any of reconfiguration actions (addition, removal or replacement of components) without taking the system offline, while ensuring consistency. (2) In this, bindings are created dynamically by discovering the service(s) and binding these services to application based on context. We focus on a set of scenarios that require the support of application reconfiguration, which is achieved by the latter.

In mobile and pervasive systems the application behaviour can be adapted through application reconfiguration, in order to provide a more relevant service with respect to changed location information (for example, when the user changes location from a shopping mall to a cinema, she should be provided with the service, which enables the user to access information about movies being exhibited in that cinema) or to deal with resource variability (e.g. changing the component that sends video information with a component that provides a text version of the video in response to a large drop in bandwidth). Other examples may be changing a filter component that forwards all messages to the user with one that forwards only important message when the user is at a meeting or replacing the light component of room1 with the light component of room2 when the user has moved in room2.

2.4.4 Dynamic Reconfiguration of Remote Method Invocation

This adaptation approach involves intercepting remote method invocations, and redirecting or modifying these invocations dynamically as a part of dynamic adaptation. The systems that employ this adaptation technique include QUO [33,34], ACT [32] and [35]. The main idea behind these approaches is the use of interception of a remote method call and then to perform actions, such as sending a request different from original, forwarding the request with modified parameters, redirecting the request to the different target etc. The interception is achieved through a redirection that can be implemented by an interceptor component or a wrapper. For example, ACT employs an interceptor to intercept the request and a proxy component to perform adaptation, while QUO employs a wrapper (or delegate), which wraps the stub, to intercept the request and to perform adaptation.

2.4.5 Code Mobility

Code mobility involves dynamically moving program code from one location to another. This technique has been studied and applied in research domains, such as traditional distributed systems and mobile computing. Many systems [10,11,29] focusing on mobile computing employ code mobility to support or enhance service provisioning to mobile users/devices. In addition, resource-limited devices can take advantage of this approach by

dynamically downloading software components only when needed in order to save their resources. Another use of code mobility is to achieve load-balancing, where a component or service running on an overloaded machine is moved to an under-loaded machine. Distributed middlewares, such as RMI [30] and JINI [31] provide the support for code mobility in that classes and stubs are downloaded dynamically by the Java virtual Machine when not found locally. This capability allows new types and behaviours to be introduced into a remote Java Virtual Machine, thus dynamically extending the behaviour of distributed applications. This capability gives RMI and JINI a significant advantage over their counterparts CORBA [59] and DCOM [60].

2.4.6 Discussion and Summary

Each of the adaptation approaches discussed above uses a different technique and can realize powerful adaptations. Adaptations realized by each approach can fulfil the adaptation needs of a particular set of applications. The selection of which adaptation approach to use depends on what kind of adaptive applications are targeted. We mentioned and discussed examples in each adaptation category and that may help choose suitable adaptation approach(s) for targeted context-aware applications. We target a set of adaptive context-aware applications which are required to discover the service(s) and bind with them based on context (contextual reconfiguration), and the bound services modify their behaviour in response to context (contextual adaptation). We combine both application reconfiguration to realize contextual reconfiguration and component/service reconfiguration to realize contextual adaptation, and put forward a system, which provides runtime support for supporting both context-aware features.

2.5 Reconfiguration to manage bindings

Our primary focus is to provide runtime support for both contextual reconfiguration and contextual adaptation in order to provide a broad scope of adaptation, and to provide a high-level means to simplify the task of developing and modifying adaptive context-aware applications. However, various failure conditions can arise during the execution of such applications, making bindings between application and bound services invalid. To this end we propose and implement a simple design approach to reconfiguration to manage bindings. An application may fail to function correctly due to failures in locating the required service during a reconfiguration process; or when the reconfiguration process successfully discovered a service and created a binding between the application and the found service based on context, an interaction between an application component and the bound service can be affected due to

network-induced problems, such as time-outs, temporary network failure or power failure at the hosting device where the bound service is running. When these kinds of network-induced problems occur, remote exceptions are generated. These are traditionally handled by having an exception handler at the client that would try to invoke bound service a few times, and if it does not succeed it would attempt to discover a new service. The problem of power failure at the hosting device where the bound service is running is different because the bound service becomes unavailable, causing the binding to become invalid. The binding is said to be invalid when the reference (real proxy/stub) to the bound service becomes invalid. Even if the power of the hosting device is restored and the same service is run again, the reference to the bound service obtained before the power failure is no longer valid, causing all the bindings to this service to become invalid. Any interaction with the service through an invalid stub would result in a remote exception being generated.

There are other situations that may cause bindings to become invalid, for an instance, when the bound service is moved to another location over a network for load-balancing purposes, or it has been moved closer to an entity accessing that bound service in order to improve service provisioning or to save bandwidth. In these situations a reference to the bound service becomes invalid upon its migration to a new location, causing all the bindings between software components and the moved service to become invalid. The above discussion shows that there are two primary causes of invalidating a binding: one is due to power failure at the hosting device where the bound service is running and other is the migration of a bound service to a new location. The solution to this problem requires updating the invalid reference to the bound service. This problem can be solved by any of two design approaches: (1) the client itself must be responsible for handling this issue. When the bound service becomes unavailable or is migrated, the old reference to this service is no longer valid and if the client uses invalid reference to this service, an exception will be thrown. The client has to handle the exception, for example, updating the reference and repeating the call. (2) The system must be responsible for handling this issue where updating a reference is carried out by the system. In the former, the burden is on the client and such reconfiguration is not application transparent, while in the latter it is done by the system and such a reconfiguration is application transparent.

There are various research efforts focusing on this issue and provide system level approach to maintain a reference with a moving object, for an instance, [62,63]. Other systems that provide application transparent support for managing references upon component migration or replacement include [47,64]. There are other research efforts that provide high-level programming model for developing adaptive context-aware applications also consider failure conditions causing binding to become invalid, and these include [61,69]. We advocate an

application transparent reconfiguration to manage invalid bindings and suggest a simple design approach, in which an application component communicates with a bound service not directly by using a real stub of the bound service, but by using a virtual stub. We discuss our design approach to managing bindings in chapter 3.

2.6 Policies and Context-Awareness

2.6.1 Background

A policy can be defined as *"a rule that defines a choice in the behaviour of a system"* [27]. Policies can take the following rule-based format as suggested by [65]:

IF { condition(s) }
THEN { action(s) }.

This means that if the condition is true, an action is taken. One of the main aspects of context-aware adaptation is a process of making an adaptation decision, which is when to perform what adaptation. Based on this observation, it can be seen that there is notional equivalence between the definition of a policy and the adaptive context-aware application. To help convey this meaning, we can use a scenario in which we consider a simple context-aware application that controls an air-conditioning unit (AC) that is activated when the ambient room temperature rises above a certain level. This scenario can be represented through an Event-Condition-Action (ECA) policy. Different policy systems provide different syntax to express an ECA policy, although similar structure. For example, in Ponder2 [26,80,94], above scenario can be represented through an ECA policy in figure 2.1.

```
1. policy := root/factory/ecapolicy create.
2. policy event: root/event/tempevent.
3. policy condition: [ :temp |(temp > 20)].
4. policy action:[ tempnontroller on.]
…….
…….
```

Figure 2.1: Demonstration of an ECA policy

The above ECA policy says that when a temperature event occurs (line 2) and the room temperature is above 20 (line 3), an AC unit should be turned on (line 4).

As can be noted, the policy description is a suitable way to express or define a context aware application. Policy-based approaches have been used to support dynamic adaptation in

various fields. For example, polices have been exploited in network and system management to define context-awareness (how the system should adapt in response to context, such as failures or change in application requirements etc.). By specifying context-awareness (adaptation) separately and through a policy specifications provides a separation of concerns between adaptation concerns and the rest of the system. This separation of concerns provides an opportunity to dynamically change the adaptation policies without changing the implementation and interrupting the system.

Policy-based techniques have also been used in middlewares, such as Services with Context-awareness and Location-awareness for Data Environments (SCaLaDE) [10] and Context-and Location-based Middleware for Binding Adaptation (Colomba) [51]. These middleware focus on mobile environments and support mobility-enabled resource bindings in addition to other things such as disconnection support, adaptation of service results to fit specific device characteristics etc. In these middlewares, a mobile terminal/mobile proxy can refer information resource through various types of binding strategies (resource movement strategy, copy movement strategy, remote reference strategy and rebind strategy [54]). The decision of which binding strategy should be used between a mobile terminal/mobile proxy and required resources when the mobile terminal/mobile proxy moves is based on deployment conditions, such as terminal capabilities (e.g., CPU power, memory space etc.) and available bandwidth etc. These binding strategy decisions are expressed explicitly through high-level policy specifications, thus providing a separation of concerns between binding management concerns and application logic. This separation of concerns reduces the complexity involved in development of mobility-enabled scenarios and allows dynamic programmability of binding strategies. Context-awareness here captures the binding management concerns where, in response to context (migration of a mobile component), a particular binding strategy is selected based on deployment conditions, and it is explicitly defined through a policy specification that has ECA (Event-Condition-Action) format.

In the field of code mobility the policy-based approach has been used to define choices regarding when, where and which components to move in response to context (e.g., user mobility, low battery power, etc.), for instance [66]. Context-awareness here captures migration strategies and is expressed explicitly through the policy specification in ECA format. In these systems, mobility concerns are explicitly expressed via high-level policy specifications that separate mobility concerns from application functionality. This separation reduces complexity involved in developing mobile-code applications, and also allows dynamic modification of mobility concerns, thus allowing dynamic reconfiguration of mobile-code applications.

Other research efforts that employ a policy-based programming approach for achieving dynamic adaptation include [12,13,18,20,67]. These systems use a policy specification in ECA format to define dynamic adaptation where the core application behaviour is modified in response to context.

Recently policy-based techniques have been applied in body sensor networks [41,42], where polices are used to define behaviours such as monitoring medical conditions of a patient (e.g., glucose level, heart-rate, etc.) and providing alerts or suggesting a course of action to the patient in event of abnormal conditions. For example, a "glucose high" policy may be defined to detect the glucose level of the patient and inject an appropriate dose of insulin when glucose level crosses, for example, 180.

The above discussion shows that the policy-based approach is an effective means of supporting context-awareness as adaptation concerns are captured declaratively in ECA rules as explicit policies. Adaptation concerns may be in the form of binding management concerns, migration concerns, network or system management concerns or core application behaviour modification. As policies are explicit components, they provide a separation of concerns between adaptation concerns and application logic. This separation of concerns and the declarative nature of policies offer these benefits: (1) it reduces the complexity involved in developing adaptive context-aware applications and (2) it allows modification of adaptation concerns (thus extending and modifying the applications) without changing the application code and interrupting the system.

2.6.2 Policy Systems

In this section, we review some popular existing policy systems that have been used in various domains, such as system and network management, mobile and pervasive computing.

2.6.2.1 Ponder

The Ponder policy language [27], developed at Imperial College London, is a declarative, object-oriented programming language that can be used to specify both security and management policies for distributed object systems. Ponder supports four policy types: obligation, authorization, refrain and delegation policies. The basic design of Ponder policies involves subject and target. The subject can be seen as an access controller or adaptation controller depending on what it controls, and target as an entity that can model resources or service providers. The Ponder supports a Domain idea. Domains provide a means of grouping objects to which policies apply and can be used to partition the objects in a large system according to geographical boundaries, object type, responsibility and authority. Obligation

policies are event-triggered condition-action rules that can be used to specify what actions the subject must perform on the target when a certain event occurs. In system management, obligation policies can be used to define behaviours, such as when to perform backups, register new users in the system, or install new software etc. In mobile computing the obligation policies can be used to define binding management concerns (e.g., which binding strategy to apply in response to mobile entity migration) and mobility concerns (e.g., which component to move when and where in response to context). Furthermore, in pervasive computing scenarios the obligation policies can be used to define reconfiguration directives which involve creating bindings in response to context, and adaptation concerns that involve modifying the behaviour of the system through parameter adjustments in response to context.

Authorization polices are used to specify which resources or services (target objects) a subject (e.g., management agent, user etc.) can access. A positive authorization policy is used to define the actions the subject is permitted to perform on target objects, while a negative authorization policy specifies the actions the subject is forbidden to perform on target objects.

Refrain policies are used to define what actions a subject is not permitted to invoke. Refrain policies are similar to negative authorization, but the main difference between the two is that refrain policies are enforced by a subject, while negative authorization policies are enforced by a target object. Refrains are used where a target is not trusted. Delegation policies are used to permit the subjects of an authorization policy to delegate some or all of their access rights to new subjects.

2.6.2.2 PDL

The Policy Description Language (PDL) is a declarative policy language [68]. Policies in PDL take the event-condition-action rule format and are similar to Ponder obligation policies. However, PDL does not support authorization policies. The policy rules defined in PDL has the following format:

event causes action if condition

The above obligation policy format says that when an event occurs and the condition is true the action should be performed. The actions may be in the form of a local call or a remote method call.

2.6.2.3 Ponder2

Ponder2 [26,80,94] is a re-design and re-implementation of Ponder, which is used by many in both academia and industry. Ponder2 is a light-weight, self-contained, extensible and

scalable policy system for specifying and enforcing policies, and can be used at different levels of scale from small resource-constrained devices (e.g., PDA and mobile phones) to complex environments.

Ponder2 is implemented using the concept of a Self Managed Cell (SMC) [102]. An SMC is an architectural pattern for implementing ubiquitous systems which require self-configuration and adaptation in response to changing conditions. An SMC allows grouping a set of hardware and software components to form systems / domains capable of configuring and adapting themselves in response to changing conditions, e.g., device failures, availability or loss of services, change in context (e.g., user activity, clinical conditions of patients in e-health scenarios, etc). These autonomous systems are called SMCs. The SMC's core design includes a publish/subscribe event notification system, which is used by the services within the SMC to interact with each other, although other forms of communication (e.g., RPC) can take place. For example, a context monitor service (e.g., timer) generates an event and the policy responds to this event, and the policy action may use remote procedure call on the managed resource (e.g., light service). The event system forwards events produced by the services (e.g., monitors such as a timer) to interested parties (e.g., policies) within the SMC who have subscribed to receive the event. The event-based interaction method provides a decoupling between the services as an event producer does not need to know about the receivers of that event, thus providing the flexibility that new services can be added to the SMC without disturbing the behavior of existing services. Another SMC's core design component is the policy service, which provides an infrastructure within the SMC for specifying and enforcing policies for self-configuration and adaptation. To summarize, SMC's support for self-configuration and adaptation is achieved through the use of an event system in combination with a policy-based service.

Ponder2 SMC allows specifying and enforcing policies, grouping components of a SMC in domains for management purposes and also provides the ability to dynamically load new functionality into the SMC. Policies can be written to control various functions within the SMC. For example, in the domestic environment the SMC may include various services (e.g., the light service, TV service, air-conditioning service, etc) and the binding policy may involve discovering various services and binding to them based on location of the user and other search criteria; the adaptation policy may involve modifying the behavior of the service (e.g., the light service) in response to user activity context; the management policy may involve adding and removing users from the environment; policies may be defined for the management (i.e., loading, removal, activation) of the policies themselves. Ponder2 SMC's core services that constitute the functionality of Ponder2 SMC include a policy service, an event service and a domain service, and Ponder2 SMC is instantiated by starting these services. Ubiquitous systems

which have been implemented using the concept of the SMC include body-sensor networks [41, 42] and unmanned autonomous vehicles [103].

Ponder2 is a general-purpose object management system and comprises *Events, Policies, an Obligation Policy Interpreter, Authorization Policy Interpreter, Command Interpreter and Domain Service*. The *Domain Service* provides a means of grouping *managed objects* (we will discuss this later) into a hierarchical structure for managing objects. The *Obligation Policy Interpreter* handles Event, Condition and Action rules, while the *Authorization Policy Interpreter* caters for both positive and negative authorization policies. The *Command Interpreter* accepts commands written in a high-level configuration language called *PonderTalk*, which may perform invocations on managed objects (we discuss *PonderTalk* later).

A *managed object* is an entity in Ponder2 capable of receiving and replying to PonderTalk message keywords, and can also receive messages from within and send them to other managed objects. The *managed object* is written in Java and its methods can be annotated (i.e. *@Ponder2op ("performAdaptation* :")) to establish the links between *PonderTalk* message keywords and Java methods. A Java method in a *managed object* is annotated as above can be called via the *performAdaptation* PonderTalk message keyword. *Managed objects* can be used to represent monitors (e.g., user presence, timer, and activity), local services (e.g., light service, temperature controller service, and alarm clock service) and adaptors or proxies to external remote objects.

Ponder2 has several pre-defined managed objects such *as Policy (obligation and authorization), Event, and Domain*. Policy-based systems based on Ponder2 may require developing other managed objects and those may be created in the Java language by following the Ponder2 requirement for any java object to be *a managed object* (e.g. implementing the *ManagedObject* interface).

Ponder2 uses a *factory object* concept to create instances of a *managed object*. Ponder2 has built-in *factory objects* for all pre-defined *managed objects* and these are created through their corresponding *factory objects*. For any other *managed object*, a *factory object* needs to be created and then its instances can be created through that *factory object*. The Ponder2 system provides a means (i.e. *PonderTalk load* keyword) for creating a *factory object* for a *managed object* in that the *managed object* is loaded into memory and its corresponding *factory object* is created. This is the same as any object-oriented system where a class has to be loaded before its instances are created. Creating the instances of *managed object* through its corresponding *factory object* provides Ponder2 with the ability to load all the code needed on demand, thus making Ponder2 extensible.

As discussed above that one of the core services of Ponder2 is an event service and services within Ponder2 SMC interacts with each other by means of an event. An event is used to disseminate contextual information such as user presence, environmental temperature, light etc. into the Ponder2 environment. An event of a particular type can be created and associated with a monitor where the monitor interacts with physical sensors to obtain contextual data and pass it to the event. Once the event has received contextual data, polices that have subscribed to that event would get triggered and perform adaption as dictated by the action part of the policies when a condition is true. For example, the user presence event can be created by sending the *event factory* a create command with an array of attribute names (user and location) that the user presence event will carry (*upevent: = root/factory/event create: #("user" "location").*) and be given to a UserPresenceMonitor.

The Ponder2 policy system currently supports two policy types: *Obligation Policies* and *Authorization Policies***.** *Obligation polices* are Event-Condition-Rules(ECA) that specify what actions must be performed when an event occurs and condition is true, while *authorization policies* specify what actions must be performed when a condition is true to protect the resources and services from unauthorized access. *Obligation policies* are created by sending the policy factory a create command (e.g., *policy: = root/factory/ecapolicycreate*.).

The Ponder2 system has a PonderTalk language, a high-level configuration language and is used to control and configure the Ponder2 system. In Ponder2 everything is a *managed object, and managed objects* make up a Ponder2 System. All *managed objects,* including Ponder2's predefined (Policy, Event, Domain) and others a developer writes for developing Ponder2-based systems, are instantiated, controlled and interacted with through PonderTalk. Generally, in all policy languages policies are specified declaratively, thus reducing the complexity involved in developing adaptive context-aware applications using policies. However, the *PonderTalk* provides higher-level abstractions to specify policies for adaptive context-aware applications and this provides better user transparency.

2.6.3 Choosing a policy system

In section 2.6.1, we discussed the use of policy-based approaches in the literature to define context-awareness in event-condition-rule format as a separate, explicit policy component, and the advantages offered by this approach. In section 2.6.2, we reviewed three successful policy systems, Ponder, PDL and Ponder2, and it can be noted that all support the ECA model, thus fulfilling the basic requirement for supporting context-awareness.

Although as discussed before Ponder and PDL make use of ECA rules for adaptation, these are primarily aimed at the management of large distributed systems and network elements and don't scale down to small devices. In contrast to Ponder and PDL, Ponder2 is a light-weight, self-contained and extensible policy system that can be used at different levels of scale from small resource-constrained devices (e.g., PDA and mobile phones) to complex environments. In addition, although the Ponder and PDL systems provide high-level abstractions to specify ECA rules, a developer is still required to know low-level details of reconfiguration actions. However, Ponder2 uses a higher-level configuration language called PonderTalk to specify event-condition-action rules. The use of PonderTalk provides better transparency to the developers in the sense that reconfiguration actions in policy specifications have to be specified at higher-level of abstractions.

Together with its ability to be used in the environments ranging from small devices to distributed systems and the use of PonderTalk for policy specifications make Ponder2 an ideal choice as a Policy system. Based on the above observations we chose the Ponder2 policy system and build our system on top of it.

2.7 Summary

We have reviewed features of context-awareness defined and discussed by various researchers, and argued that both contextual reconfiguration and contextual adaptation are more interesting context-aware features as these involve adapting the application behaviour based on context, thus helping to eliminate unnecessary user cooperation and making technology as "calm" as possible. We have discussed and argued that applications belonging to these two context-aware features can be considered to be adaptive context-aware applications and are a subset of context-aware applications, but not vice versa. We have mentioned various research efforts focusing on contextual adaptation and contextual reconfiguration individually. However, we envision many adaptive context-aware application scenarios that involve both context-aware features and thus we focus on combining both.

We have gone on to review and discuss various adaptation approaches that have been used in the literature to realize dynamic adaptation. Selection of which adaptation approach to use depends on what kind of applications are targeted. In order to support both contextual reconfiguration and contextual adaptation we combine both the application reconfiguration approach to realize contextual reconfiguration and component/service reconfiguration to realize contextual adaptation.

An important issue related to providing support for adaptive context-aware applications is that of considering failure conditions that can arise due to network problems and migration of software component to some other location over the network. We have briefly discussed these conditions and our application transparent reconfiguration approach to address this issue, and compared it with other approaches. We have made an argument that providing reconfiguration support to manage bindings at system level relieves a developer from the task of handling such problems and enables her to focus on real requirements of adaptive context-aware applications, thus contributing towards simplifying the task for developing adaptive context-aware applications.

We have provided an overview of policy-based approaches for supporting context-awareness and argued that the policy-based approach is an effective means for supporting context-awareness as adaptation concerns are captured declaratively in ECA rule as explicit components, thus providing separation of concerns between adaptation concerns and application logic. Benefits arising out of the separation of concerns and declarative nature of policies are a reduction in complexity involved in developing adaptive context-aware applications and allowing modification of adaptation concerns without changing application code and interrupting the system.

We have reviewed three successful policy systems, Ponder, PDL and Ponder2, and have noted that all three support ECA policies, thereby satisfying the basic requirement for supporting context-awareness. However, out of these three, we have chosen Ponder2 to build our system supporting contextual reconfiguration and contextual adaptation on top of it due to these reasons— that it can be used at different levels of scale from small resource-constrained devices (e.g., PDA and mobile phones) to complex environments, and it uses the PonderTalk configuration language for policy specifications. PonderTalk provides better transparency to developers in that reconfiguration actions in policy specifications have to be specified at higher-level of abstractions.

To provide a broad scope of adaptation and to simplify the task of developing and modifying adaptive context-aware applications, we combine contextual reconfiguration and contextual adaptation and provide runtime support for both, and provide a policy-based programming technique using Ponder2. We have built our system, PCRA on top of Ponder2, thus providing policy-based support for contextual reconfiguration and contextual adaptation. Ponder2 provides basic support for policy specifications and triggering of policies in response to context, while our reconfiguration and adaptation infrastructure within PCRA provides broad runtime adaptation support. Moreover, we have integrated application transparent reconfiguration support in PCRA to handle the issue of managing bindings. We have also

integrated caching support in PCRA for improved performance, where virtual stubs holding references to real proxies are cached. When establishing a binding between an application component and a remote service in response to context, the system gets a virtual stub for that remote service from a cache if available and gives it to the application component without doing lookup. We have not come across any research efforts in the literature that supports both context-aware features (contextual reconfiguration and contextual adaptation) where both forms of adaptation are controlled with policies using a specialized policy language (e.g., Ponder2), and also considers failure conditions causing bindings to become invalid and provide application transparent reconfiguration support to address this issue.

Chapter 3

Supporting Contextual Reconfiguration and Adaptation

In the previous chapter, we have established an argument of this thesis in which we combine both contextual reconfiguration and contextual adaptation, two adaptive features of context-awareness, to provide a broader scope of adaptation, and the use of a policy-based approach as a programming model to simplify the task of developing, modifying and extending adaptive context-aware applications. As a proof of concept, we have designed and implemented a system, PCRA and various hypothetical adaptive context-aware scenarios to support the main arguments of this thesis. As discussed in the previous chapter, there are other important surrounding issues that we address which are associated with adaptive context-aware applications. These include managing invalid bindings, providing caching support and personal conflicts. The bindings may become invalid for various reasons, such as non-availability of the services due to power failure at the hosting device where the bound service is running, or the bound service being moved or replaced. We provide caching support of virtual stubs in order to improve system performance where virtual stubs holding references to real proxies are cached. A user conflict issue arises when there are multiple users in the environment with a different set of preferences for different services. We would like to mention here that our main focus is not addressing the issue of user conflicts, which is a separate, vast and interesting research area. There are various research efforts such as [73-75] focusing on this research issue. However, we embed a simple solution to this problem in the system, which is priority-based— the older the user, the higher the priority. We will discuss this briefly in chapter 5 in which we provide implementation details of the proposed system through an example scenario that involves multiple users. In this chapter, we present a high-level system architecture of PCRA and discuss briefly its main elements, and then discuss how contextual reconfiguration, contextual adaptation, reconfiguration to managing bindings and caching support are supported within the system.

3.1 System Architecture of PCRA

In the following sections, we discuss the high-level concepts of each contribution using system diagrams in which the interaction of components is presented and described. The overall

architecture of PCRA that provides the support for all the contributions is comprised of three parts as shown below in figure 3.1: the Ponder 2 System, our reconfiguration and adaptation infrastructure and Java RMI.

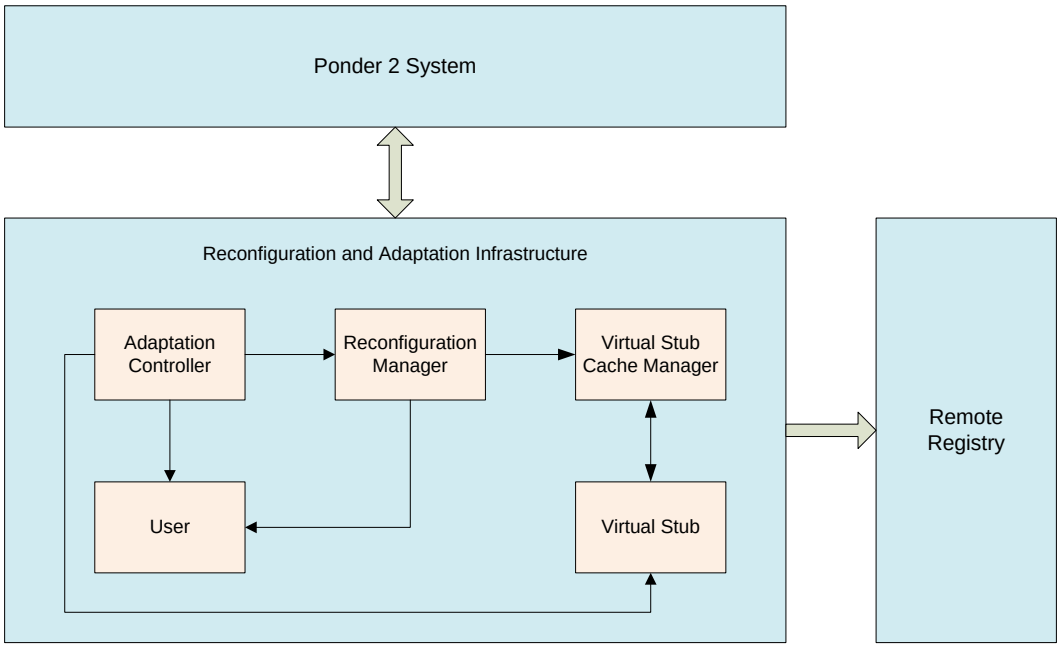

Figure 3.1: High-level System Architecture of PCRA

We have briefly discussed the Ponder2 system in chapter 2. It is a general-purpose object management system and provides the support for both obligation policies and authorization policies, and has Events, Policies, an Obligation Policy Interpreter, Authorization Policy Interpreter, Command Interpreter and Domain Service. Our reconfiguration and adaptation infrastructure provides runtime support for contextual reconfiguration, contextual adaptation, reconfiguration to manage invalid bindings and seamless support for virtual stubs, as discussed previously. RMI is a distributed middleware that provides the means for developing remote services and remote method communication between these services and their clients. All three parts of the overall architecture work together to provide the system, PCRA, in which adaptive context-aware applications can be developed using polices. As can be noted from above discussion, PCRA architecture provides infrastructure for specifying and enforcing policies via Ponder2 system, and also provides a broader support of adaptation (contextual reconfiguration, contextual adaptation and reconfiguration support to recover from invalid bindings) through our reconfiguration and adaptation infrastructure. All the developer or end user is required to do for developing adaptive context-aware application is to express binding policies and adaptation policies in Ponder2 specifications.

3.1.1 Design considerations of PCRA system

The current design of PCRA involves local communication between Ponder2 system and reconfiguration and adaptation Infrastructure and hence both have to reside on the same machine. As a result, PCRA does not allow distribution of PCRA components (i.e. Ponder2 system and reconfiguration and adaptation infrastructure) across multiple machines and thus the reconfiguration and adaptation infrastructure can not be instantiated over multiple machines, hence the centralized PCRA architecture. An instantiation of PCRA system would result in an instantiation of Ponder2 system and an instantiation of reconfiguration and adaptation infrastructure, which means one instance of PCRA comprises of one instance of Ponder2 system and one instance of reconfiguration and adaptation infrastructure. The one instance of PCRA provides the overall support for policy-based reconfiguration and adaptation infrastructure, where Ponder2 instance supports specification and enforcement of policies while reconfiguration and adaptation infrastructure instance, in addition to other functionality, is responsible for establishing bindings to various services in the environment in response to policy evaluation. As reconfiguration and adaptation infrastructure can not be instantiated across multiple machines, there would be just one instance of reconfiguration and adaptation infrastructure available to provide the support for reconfiguration and adaptation.

The one instance of reconfiguration and adaptation infrastructure may suffice in the pervasive computing environments, such as domestic environments where the number of bindings required to different services is assumed to be reasonably smaller. However in other environments such as an airport and large-sized offices the scalability issue would come up and require considering the distributed design of PCRA. In the distributed design of PCRA, the reconfiguration and adaptation infrastructure would be instantiated across multiple machines and the responsibility of creation of bindings would be distributed to different instances of reconfiguration and adaptation infrastructure, thus improving the scalability of PCRA. However, there are challenges and trade-offs involved in the distributed design of PCRA, which may include:

- The reconfiguration and adaptation infrastructure will work as a remote system, where its system components (reconfiguration manager, adaptation controller and user) will be remote services. In order for Ponder2 to interact with remote reconfiguration and adaptation infrastructure there would require implementing corresponding adapters for reconfiguration manager, adaptation controller and use. Moreover, distributed design of PCRA would require system component(s) for monitoring load on various instances of the reconfiguration and adaptation infrastructure and distributing the request for creation of bindings to them, hence additional code required.

- Ponder2 system would interact with the reconfiguration and adaptation infrastructure through RPC (remote procedure calls), and a remote method call is a far slower than a local call, thus affecting the system performance.

We have chosen the centralized design of PCRA as we target domestic environments and, as discussed before, the number of bindings required to various services is assumed to be smaller. Thus, the one instance of reconfiguration and adaptation infrastructure will reasonably be sufficient. Moreover, Ponder2 would interact with reconfiguration and adaptation infrastructure via local calls unlike via RPCs in distributed design of PCRA, hence improved system performance.

3.2 Contextual Reconfiguration

As discussed in chapter 2, contextual reconfiguration is one of the features of context-awareness and that applications under this category can be considered to be adaptive context-aware applications in the sense that the application behaviour is adapted by discovering a service based on context and then binding with it to realize new behaviour. In this section, we discuss our approach to contextual reconfiguration. The approach is a policy-based in which a policy is specified declaratively in an ECA (Event-Condition-Action) format. Policies are written to respond to context events. When an event occurs and the condition is true, the appropriate policy will be triggered. This will cause the system to establish bindings between application components and the remote service(s) as dictated by the action part of the policy. We take one of the applications scenarios we have implemented and demonstrate our contextual reconfiguration approach through this.

Consider the scenario that when a user enters a particular room, say room1, the system turns ON the light in the room1 and sets the light level to its last used level. The system also turns ON the air-conditioning unit in room1 and sets the air-conditioning setting to its last used value. Furthermore, the system turns ON the light in adjoining rooms (e.g., room2) and set their light level to some low value. If the user stays in the room1 for some time (e.g., for a minute), the light level in that room1 is set to the user-preferred value for the light and air-conditioning setting is set to the user-preferred value for air-conditioning, and the lights in adjoining rooms (e.g., room2) are turned OFF.

In this scenario the context involved is a user presence and the bindings that system needs to establish would be to a light service in room1, air-conditioning service in room1 and the light service in room2. Our system design includes the user component, the architectural component that models the user of the environment and includes other functionalities, such as

allowing the users to set their preferences for different services. We discuss the user component in detail in chapter 5. Based on which user is in the environment, the system will create an instance of that user, and based on the name of services, contextual information and other search information, the system discovers the remote services and binds them to the user instance. In the above scenario, if the user, "Maanta", enters room1, the system discovers the light service in room1, the air-conditioning service in room1 and light service in the adjoining room (room2), and binds them to the user instance for Maanta. This is illustrated in figure 3.2 below.

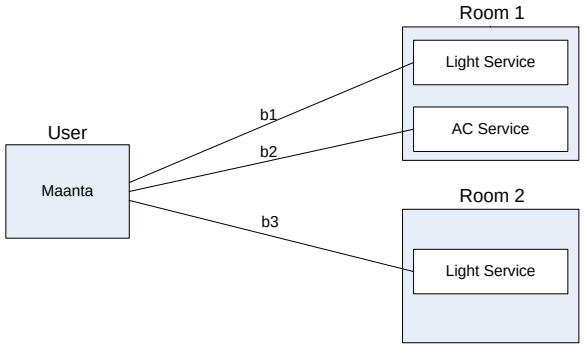

Figure 3.2**:** Bindings to the light service and air-conditioning service in room1 and the light service in room2

The implementation of this example scenario includes a binding policy and other policies. The following code snippet is the binding policy involved in this example scenario, and the complete code listing of this scenario can be found in chapter 4.

```
// A binding policy

1.  policy := root/factory/ecapolicy create.
2.  policy event: root/event/upevent.
3.  policy condition: [:user :location :enter_or_left|
         (enter_or_left == "enter")].
4.  policy action:[:user :location |
5.       theUser := users at: user.
6.       configurator createBinding: #(theUser location "LightService" "ACService").
7.       configurator createBinding: #(theUser "room2" "LightService" "otherLocation").
         ……
         ……
         ……
    ].

8.  root/policy at: "bindingpolicy" put: policy.
9.  policy active: true.
```

Figure 3.3: The binding policy in the light and air-conditioning example

The binding policy subscribes to the user presence context event (line 2 in figure 3.3) and its action part includes reconfiguration messages (line 6 and line 7), which provide means to interact with one of the system components called the reconfiguration manager. This component is in charge of performing reconfiguration. When the user presence context occurs, the binding

policy is triggered and reconfiguration messages cause the reconfiguration manager to create bindings between the user and the remote services as shown in figure 3.2.

There are other system components that the reconfiguration manager interacts with to realize the overall process of policy-based reconfiguration: the virtual stub and virtual stub cache manager. Although these components are core design components for other features that the system supports, such as reconfiguration for managing bindings and providing caching support for bindings and hence will be discussed in the sections where we discuss these features, they are involved in the overall process of reconfiguration and therefore we introduce them here. The virtual stub wraps the real proxy of the service, thereby adding a level of indirection to method calls on the actual service. While adapting the behaviour of the service based on context, the system does not directly communicate with the service, but through its corresponding virtual stub. The virtual stub cache manager is responsible for performing RMI lookups for discovering services, creating and initializing virtual stub instances and caching them. We discuss in section 3.3 our approach to caching virtual stubs for improved performance. Having introduced these two components, we discuss the steps and the interaction involved between the reconfiguration manager and these components in carrying out contextual reconfiguration.

- When the user presence context event occurs (the user being detected in a specific location—this context includes two attributes: the user ID and location), the binding policy is triggered and its action part is executed. The first message (line 5) in action part is reproduced below.

  theUser: = users at: user.

  The *users* in the above message is a domain, which was created when the PCRA system was started and run (we will return to this in chapter 5), contains various user instances. The user instances are created, added to and removed from the *users* domain dynamically through a GUI-based system utility that we have implemented (again discussed in chapter 5). The *user* in above message is a user ID attribute coming from user presence context event. The *at:* message takes *user* as an attribute and returns the user instance (*theUser*) for the *user*. For example, if the *user* attribute of the user presence context has the value "Maanta", the user instance for "Maanta" will be returned. The other two messages (line 6 and 7) in action part of the binding policy are reconfiguration messages. The first reconfiguration message (line 6) has four attributes: (1) *theUser*, the user instance returned (line 5), (2) *location* attribute of the user presence context event, (3) and (4) are the service names (*LightService* and *ACService*). This reconfiguration message is mapped into a method call to the *reconfiguration manager,* and this method call discovers the light service and air-

conditioning service in room1 (location = “room1”) and binds them to the user instance (*theUser*). Similarly, the second reconfiguration message (line 7) causes reconfiguration manager to discover the light service in room2 and bind it to the same user instance. The second argument of the reconfiguration message generally indicates both the current location of the user and also the location of the services. However, the second reconfiguration message (line 7), which is “room2” only indicates the location of the light service as informed by the last argument of this message (“otherLocation”). This is tag information to reconfiguration manager to consider the second argument as related to location of the services, not current user location. The following steps discuss which other components the reconfiguration manager interact to establish bindings between the user instance and remote services.

- The *reconfiguration manager* communicates with the *virtual stub cache manager* and asks for the *virtual stub* for each of the services (the light service for the room1, air-conditioning service for room1 and the light service for room2).
- The *virtual stub cache manager* searches its caches to determine if it has the required *virtual stubs*. If available, it hands them to the *reconfiguration manager* and then the *reconfiguration manager* delivers them to the user instance. If not found in the cache, it performs a lookup to discover the real proxy for each of the services and cache them.
- In order to discover the remote services, the *virtual stub cache manager* communicates with the RMI remote registry and performs the following actions:
- It discovers the real proxy for each of the remote services,
- It creates an instance of a *virtual stub* for each of the services and initializes it with the corresponding real proxy, and now each of the *virtual stubs* wraps/holds the real proxy of the corresponding remote service. Now there is a *virtual stub* for the light service in room1, a *virtual stub* for the air-conditioning in the room1 and a *virtual stub* for the light service in room2.
- It caches all these *virtual stubs* and then hands them to the *reconfiguration manager*.
- When the *reconfiguration manager* has received the *virtual stubs*, it hands them to the user instance. Once the user instance has the *virtual stubs*, this means that the bindings have been created between the user and the remote services, and now the user has bindings with the light service in room1, air-conditioning service in the room1 and the light service in the room2. Figure 3.4 shows policy-based reconfiguration process.

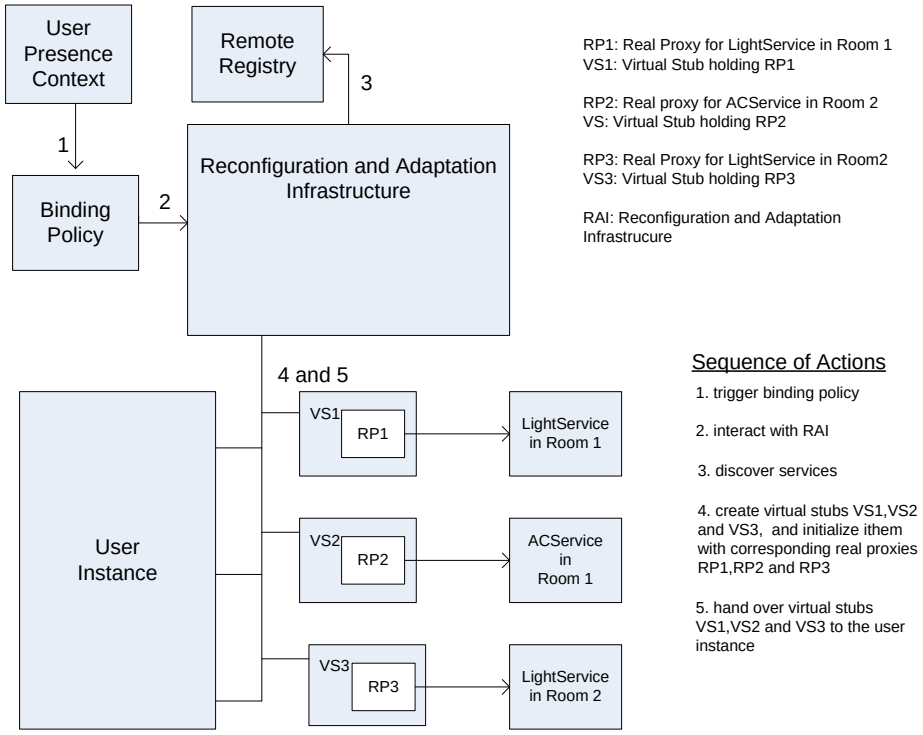

Figure 3.4: Policy-based reconfiguration

3.3 Caching support for improved performance

One of the features of the proposed system, as discussed before is the seamless caching support of virtual stubs in order to improve system performance. As discussed in section 3.2, the virtual stub holds a reference to the real proxy of the remote service and the user has a binding with the remote service through the virtual stub. If we look at the binding process, one of the steps involved is performing a remote lookup to discover the remote service in response to context. This remote lookup process provides the largest contribution to the overall binding time and this is due to the fact that the remote method calls are much slower than local calls, at least 1000 times slower [58]. In adaptive context-aware applications, it turns out to be undesirable in terms of user experience. For example, in the example scenario, introduced in section 3.2, the user experience may get affected when the user enters room1 and light or air-conditioning in room1 is not turned on and adjusted to user's preferred value immediately. In the context of other distributed systems, this may affect other applications running on the network as every remote method call decreases the amount of bandwidth available on the network for all the applications using the network. To this end we propose and implement a seamless caching technique to improve system performance in which virtual stubs are cached locally and when these are required during reconfiguration, these are obtained locally if at all possible. During the

binding process when the system discovers a particular remote service for first time, it takes longer since it involves a remote lookup. When the binding to the same service is required again, the system takes less time since the service is obtained from the local cache directly without the need for a remote call. For example, in our scenario, when the first user enters room1, the binding time taken by the system would be far higher as the light service and air-conditioning services are discovered by performing remote lookups. When this user leaves room1, the light service and air-conditioning services are turned off and after some time other user enters the room1, the system would create the bindings between the user instance for the new user and both the light service and air-conditioning in the room1, and the binding time taken by the system would be greatly reduced in comparison to what it would be in the former case since both light service and air-conditioning services are obtained locally. In chapter 6 we show that the caching technique significantly reduces reconfiguration time and hence improves user experience.

The design component in charge of providing support for seamless caching of *virtual stubs* is the *virtual stub cache manager*. We briefly introduced the actions the *virtual stub cache manager* performs in section 3.2; here these are discussed in more detail.

- The *reconfiguration manager* communicates with the *virtual stub cache manger* by invoking *getVirtualStub(serviceDescription)* and this method returns the *virtual stub*.
- The *virtual stub cache manager* has a hashtable named *virtualStub_Table* where *virtual stubs* for different services are cached. In order to return the requested *virtual stub* to the *reconfiguration manager*, it checks whether it is in *virtualStub_Table* by invoking *virtualStub_Table.get(serviceDescription).* If available, it is returned to the *reconfiguration manager*.
- If the *virtual stub cache manager* does not find it, it invokes its *doLookup(serviceDescription)* method and this, in turn, communicates with the *remote* registry to obtain the real proxy/stub of the remote service.
- After obtaining the real proxy of the service, the *virtual stub cache manager* creates an instance of the *virtual stub* and initializes it with the found proxy through *createVirtualStubInstance (serviceDescription, remoteStub)* method.
- The instance of *virtual stub* is then cached into the table through a method call *virtualStub_Table.put (serviceDescription, virtualStub)* and then returned to the *reconfiguration manager*.

The figure 3.5 below shows the interaction between the architectural components to provide seamless caching support for *virtual stubs*.

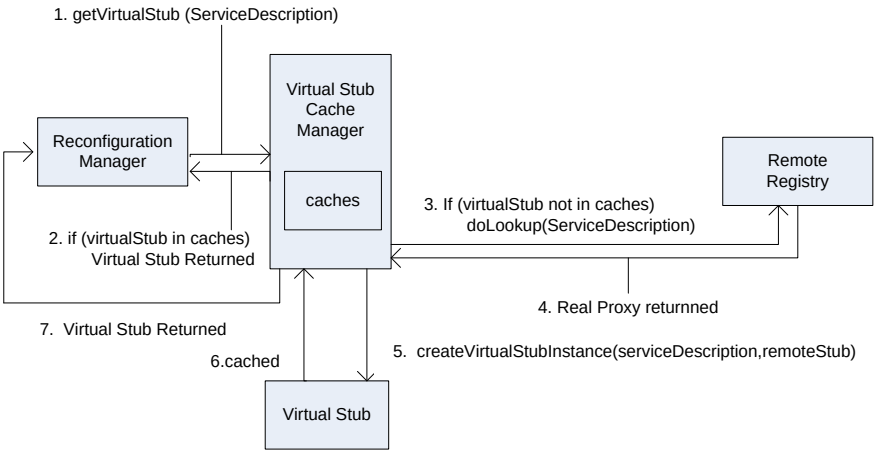

Figure 3.5: Seamless caching support of virtual stubs

3.4 Reconfiguration to manage bindings

We have described our approach to context-aware reconfiguration and caching of virtual stubs for improved performance, but there may arise situations during the execution of adaptive context-aware applications that may cause bindings to become invalid and so require reconfiguration to update them. The bindings may become invalid due to many reasons, such as sudden non-availability of the bound service (due to power failure at the hosting device where the bound service is running), or the bound service has been moved to some other location over the network for load-balancing purposes, or it has been moved closer to the entity accessing the bound service in order to save bandwidth. In all these situations, the real proxy of the bound service becomes invalid, causing all the bindings to this service to become invalid. In our approach to reconfiguration for managing bindings, when the binding becomes invalid due to any of these reasons and a method call is made on an invalid reference, an exception is thrown in the *virtual stub*. In response to the exception, the *virtual stub* immediately performs the reconfiguration to update the invalid reference and repeats the call.

The concept of the *virtual stub*, also called *smart proxy* has been used as a design component to address various issues, and some of the research efforts that make use of this component include [47,70-72]. The *virtual stub*/*smart proxy* wraps the real proxy of the remote service and provides more functionality than the real proxy does (forwarding remote calls from a client to remote service), depending on the requirements of the system. For example, it may be required to perform client-side validation before calling actual methods of a target object; it may be desirable to perform client-side caching to save the remote calls; it may be desirable that in case of any remote exception the client should not handle the exception, but instead *smart proxy*, so that the client is free to deal with real requirements of the applications, etc. Other additional responsibilities performed by the *smart proxy* may include performing security (e.g.

not giving access to certain remote objects according to IP address) and load-balancing. The use of *virtual stubs* as a design component in our system is that it is in charge of handling an exception which is generated when attempting to call the method of real proxy that it holds which has become invalid due to any reasons (for whatever reason), providing an application transparent solution to managing bindings.

The *virtual stub* in the PCRA, in addition to forwarding remote calls to the remote service, has additional responsibility of performing reconfiguration to rebind to the service by updating an invalid reference to the real proxy of the service, sending an updated copy of itself so that the *virtual stub cache manager* always has an up-to-date copy of the *virtual stub*. In response to an exception thrown as a result of a remote method call on an invalid proxy, the *virtual stub* performs the following actions:

- It invokes its *invalidreference ()*, which performs a remote lookup to obtain a new copy of the real proxy and the invalid proxy is replaced by new one, thus updating the invalid reference.
- The *virtual stub* Cache Manager still has the *virtual stub* saved which contains invalid proxy. The *virtual stub* communicates with the *virtual stub cache manager* and sends a copy of itself, which now contains the updated proxy. The *virtual stub cache manager* replaces the old copy with new one into cache.
- It then repeats the remote call.

The figure 3.6 below shows interaction between system components to support reconfiguration to manage bindings.

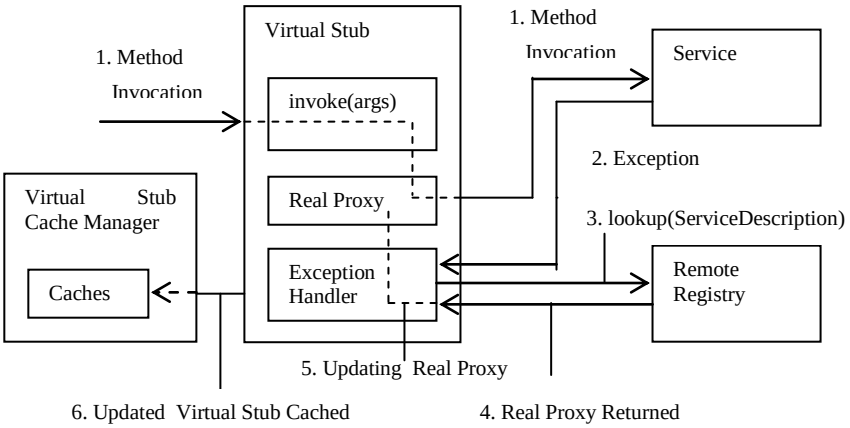

Figure 3.6: Reconfiguration to manage bindings

As invalid bindings are managed by the *virtual stub*, which is a system component, not by the application developer, this approach contributes to reducing the development efforts for developing adaptive context-aware applications.

3.5 Contextual Adaptation

We consider two separate cases in contextual adaptation as discussed in chapter 2: (a) context-triggered actions and (b) modification of these actions in response to context, hence the contextual adaptation support the system provides incorporates both. In chapter 2, we have listed and discussed various adaptation approaches used in the literature to realize the adaptation each of which enables powerful adaptations. Selection of which adaptation approach to use depends on what kind of adaptation is required. For the contextual adaptation support for modifying the actions in response to context, we employ an adaptation approach called service/component parameter adaptation in which the service/component behaviour is modified through parameter adjustments.

Our approach to contextual adaptation is policy-based where adaptations are specified through policies. The bindings between the user instance and remote service are created by policy-based reconfiguration support, and adaptation policies are specified to adapt the behaviour of the remote services involved in the bindings to satisfy the user needs, such as modifying the light value to the user's preferred value through a light value parameter, lowering down or increasing the volume of a TV/music service through a volume parameter, etc., without requiring any attention from the user. The adaptation policy subscribes to the context event and when that context event occurs, the policy gets triggered and if the condition is true, the policy performs the adaptation in terms of modifying the behaviour of the service through parameter adjustment, as dictated by the action part of the policy. We discuss our approach to contextual adaptation in detail through an example scenario which was also used to discuss our approach to contextual reconfiguration in section 3.2. The contextual adaptation involved in this scenario is that when the user stays in room1 for a minute, the light value in the room1 is modified to the user's preferred value, the air-conditioning setting is modified to user's preferred value and the light in room2 is turned OFF. The policy which implements this functionality is shown below (figure 3.7), and the complete code listing of this scenario can be found in chapter 4.

```
// A policy that if the user stays  a 1 min in the room, then switches off the light in
//adjacent room(e.g. room2), adjusts the light value and the ac setting to the user preferred values.

1.  policy := root/factory/ecapolicy create.
2.  policy event: root/event/timeEvent.
3.  policy action: [

4.      location:= configurator getCurrentLocation.
5.      theUser:= configurator getHighestPriorityUser: location.
6.      otherLocation:= configurator getOtherLocation.
7.      preferredLightValue :=theUser getPreferredValue: "LightService".
8.      preferredAcSetting  :=theUser getPreferredValue: "ACService".
9.      adaptation performAdaptation:#("LightService" location "adjust" preferredLightValue).
10.     adaptation performAdaptation:#("ACService" location "adjust" preferredAcSetting).
11.     adaptation performAdaptation:#("LightService" otherLocation "off").].
12. root/policy at: "timepolicy" put: policy.
13. policy active: true.
```

Figure 3.7: The time policy in the light and air-conditioning example

This policy subscribes to the time context event (line 2), and action part of this policy includes various messages, including three adaptation messages (line 9, 10, 11). The message (line 7) obtains the user's preferred value for the light service, and the message (line 8) obtains the user's preferred value for the air-conditioning service. These users' preferred values are already set by the user. One of the features of PCRA is that it allows the users to dynamically customize their preferences for various services through our GUI-based system utility (we discuss this in chapter 5). The time context monitor notes the time and if it has been a minute since the user had entered room1, the time context monitor would generate the time context event. In response to time context event, the time policy (figure 3.7) would get triggered and each adaptation message (line 9, 10, 11) would be mapped into a corresponding method call *void performAdaptation(P2Object args).* This one argument, i.e., args of P2Object type (we discuss P2Object in chapter 5) will hold all the attributes of the message, and this argument is converted into P2Object array and then all attributes of the message are retrieved from this array. The first *void performAdaptation(P2Object args)* call modifies light value in room1 by invoking the adjust method with parameter *preferredLightValue* (obtained through the message, line 7) , the second call modifies the air-conditioning setting in room1 by invoking the adjust method with parameter *preferredAcSetting* (obtained through the message, line 8) and the third call turns the light off in room2. The *void performAdaptation(P2Object args)* is one of the methods of an *adaptation controller*, which is one of the system components.

The *adaptation controller* has another method called *P2Object getRemoteField (P2Object args).* The *void performAdaptation(args)* and *P2Object getRemoteField(args)* methods of *adaptation controller* make a generic adaptation interface of the system (we discuss this in chapter 5). The former can be used to perform context-triggered actions or modification of these actions by performing remote method invocations on any bound service. Similarly, the latter can be used to perform remote invocations on any remote services in order to access their remote fields. As can be recalled from discussion before, the user instance holds bindings to the

remote services through their corresponding *virtual stubs*. Any adaptation through above methods on the remote service is performed through its corresponding *virtual stub*. The functionality of *void performAdaptation(P2Object args)* involves getting the required binding (i.e., the *virtual stub*) from the user and invoking the *virtual stub* method that, in turn, reflectively calls a required method of the real proxy that it holds to perform adaptation on the remote service. Similarly, the *P2Object getRemoteField(args)* involves obtaining the required binding (i.e., *virtual stub*) from the user and invoking a method of *virtual stub* that, in turn, calls the required method of the real proxy to get the remote field of the remote service. We summarize the brief discussion above into steps that capture the interaction involved between the *adaptation controller* and other system components to realize the contextual adaptation as follows:

- When the context change event occurs, the adaptation policy is triggered and adaptation messages in the action part of the policy are mapped into *performAdaptationt(args)* method calls of the *adaptation controller*.
- The *adaptation controller* through *void performAdaptation (P2Object args)* interacts with the user component and obtains the required binding (i.e., *virtual stub*).
- Once the *adaptation controller* has obtained the *virtual stub*, it interacts with the *virtual stub* and invokes the method of the *virtual stub* called *invokeMethod (args).* This method, in turn, reflectively calls the method of a real proxy to perform adaptation on the remote service.

The figure 3.8 below shows how contextual adaptation is achieved in above example scenario.

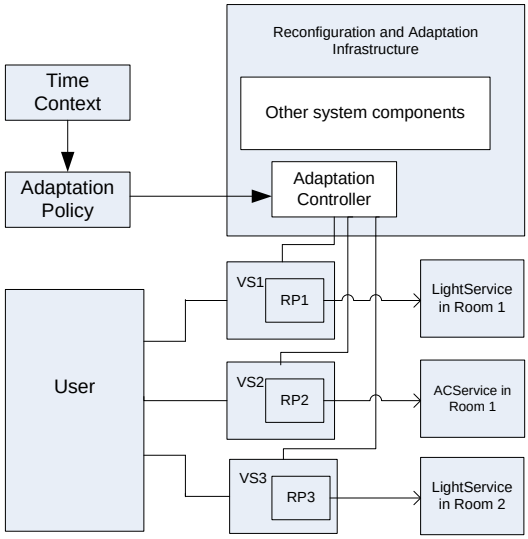

Figure 3.8: Policy-based contextual adaptation

3.6 Sequence of Messages in Policy-based Reconfiguration

The message sequence diagram below (figure 3.9) captures the sequence of messages involved in achieving policy-based reconfiguration within PCRA.

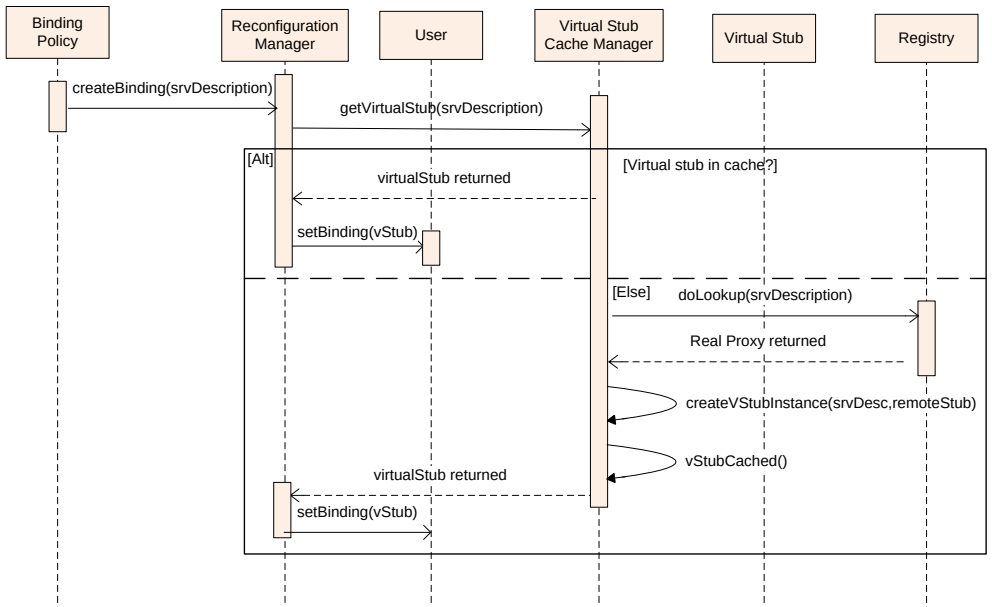

Figure 3.9: Message sequence diagram for policy-based reconfiguration

The sequence of messages in policy-based reconfiguration starts from the point the binding policy has been triggered until the establishment of the binding between the user instance and the remote service.

3.7 Sequence of Messages in Policy-based Adaptation

The diagram below (figure 3.10) captures the sequence of messages involved in achieving policy-based adaptation within PCRA.

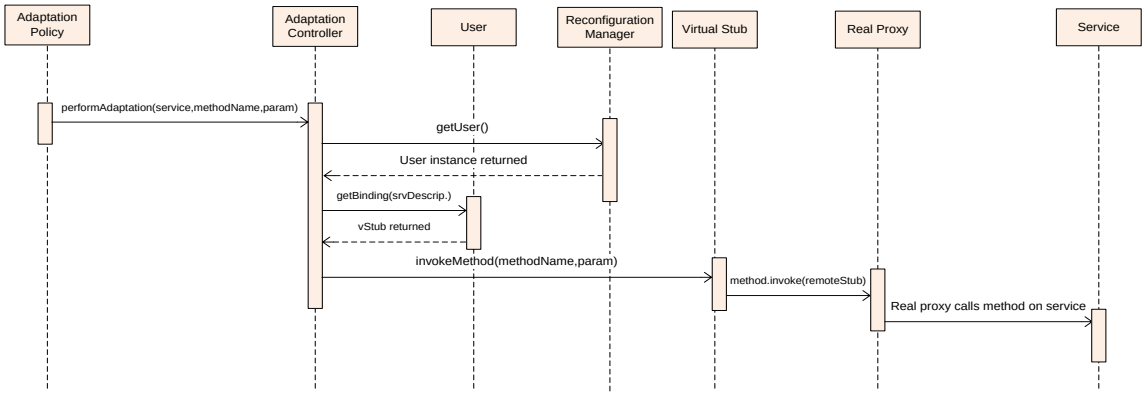

Figure 3.10: Message sequence diagram for policy-based adaptation

The sequence of messages in policy-based adaptation starts after the binding has been established, and it starts from the point the adaptation policy has been triggered until the adaptation is achieved in terms of modifying the behaviour of the service involved in the binding.

3.8 Summary

We have designed and implemented the system, PCRA that provides runtime support for our contributions: contextual reconfiguration, contextual adaptation, reconfiguration to manage invalid bindings and caching of virtual stubs for improved performance. In this chapter, we have discussed our approach to contextual reconfiguration and contextual adaptation with the help of one of the example scenarios we have implemented, and also shown architectural components involved in realizing both of these. The core feature of our approach to both contextual reconfiguration and contextual adaptation is the use of policy-based programming approach in which the binding policies and adaptation polices are specified to realize the behaviour of adaptive context-aware applications.

We also address other surrounding issues as related to providing the support for adaptive context-aware applications and these include managing bindings and caching support for virtual stubs to improve system performance. In this chapter, we have discussed our approach to these issues and also shown the system components involved and their interaction in realizing the support for these issues. We have also shown the message sequence diagram for the system which shows the sequence of messages involved in realizing policy-based contextual reconfiguration and policy-based adaptation.

Chapter 4

Hypothetical Example Scenarios

In the previous chapter, we discussed the high-level concepts behind the PCRA and our approach to achieving them through one of the example scenarios we have implemented. While discussing our approach to each of our contributions, the system components involved and their interaction in achieving each contribution was presented and described. In order to show that the proposed system provides a broader scope of adaptation (by combining and providing the support for both contextual reconfiguration and contextual adaptation) and also to justify the argument that a policy-based programming approach provides an effective means for developing, modifying and extending applications, we have implemented several hypothetical example scenarios, including all five that have been implemented in Scooby [69]. These five scenarios have been used in Scooby as the basis for comparing their approach to service composition / reconfiguration to One.World [76-78,84]. Our work and Scooby share similar research goals in the sense that we both advocate the use of high-level means to perform service composition/reconfiguration to simplify the development task. However, we use different approach to achieving this. Scooby's main idea is that a dedicated domain specific language is a more effective way of performing service composition in which composed services can be developed using high-level binding directives to discover and bind services rather than traditional approaches that use an API, whilst in contrast we advocate that the use of a policy-based programming model provides effective means for carrying out context-aware reconfiguration. In order to compare our approach to that of Scooby, we follow the same comparison methodology that Scooby has used to compare their approach to One.World by implementing five scenarios on both Scooby and One.World and using these scenarios as the basis for comparison. In addition, we extend Scooby's scenario 5 (section 4.1.2.5: the music & telephone scenario), and will show in evaluation chapter how this modification can be achieved within both the PCRA and the Scooby. In this chapter, we provide high-level description of all the example scenarios we have implemented, including all five from Scooby and also provide the code for each of the scenarios along with their description.

4.1 Prototype High-level Example Scenarios

Five scenarios have been reused from Scooby [69] and used for comparison. Three variants on these were developed in PCRA alone to best explore its features. Therefore, we divide all scenarios in two categories: PCRA high-level scenarios and Scooby high-level scenarios.

4.1.1 PCRA Example Scenarios

In this section we introduce and describe high-level example scenarios that we have implemented within our proposed system, PCRA. Each example scenario involves both contextual reconfiguration and contextual adaptation.

4.1.1.1 A Home Lighting Example Scenario

The scenario is that if a person enters the room, the system turns on the light and the light level is set to what it was when the light was last turned off. The system senses the light level and if it is less than 90 % of user-preferred value or greater than 110 % of user-preferred value, it is adjusted to the user-preferred value. The system also monitors the activity of the user (e.g., reading, sleeping, watching TV) and adjusts the light value to the user's preferred value for that activity. Further, the same process is repeated when a person is moving between rooms. We now elaborate this scenario further with an example.

Let us suppose that there is no one in room1 and now the user, "Maanta" enters room1, the user presence context widget detects her presence and sends the user presence event (user = "Maanta", location = "room1" enter_or_left= "enter"). As a response to this event, the system obtains the user instance for her from previously created user instances in the system, discovers a light service in room1 and binds it to the user instance, and then turns the light in room1 on and adjusts the light value to its last used value. The light intensity measuring widget provides data regarding the light intensity in room1 and the system responds to it as follow. It obtains her preferred value for the light from user preferences, which are saved in the system, and if the light value is less than 90% of preferred value or greater than 110% of preferred value, the light value in room1 is adjusted to her preferred value.

Now another user, "Yasir" enters room1, the user presence context widgets detects his presence and sends the user presence event (user = "Yasir" location = "room1" enter_or_left = "enter"). As a response to this event, the system gets his user instance, the light service for room1 from cache (when the light service in room1 was discovered previously, it was also cached to avoid lookup when it was needed again) and binds it to his user instance. Now users,

Maanta and Yasir are in room1 and the light value is already set to Manta's preferred value. The light intensity measuring widget senses the light intensity in room1, which is, at the moment, Maanta's preferred value and sends it to the application. In response to this, the application checks with the system about the highest-priority user (currently we have a simple personal conflicts solution based on a priority of the users and that is, the older the user, the higher the priority) and the application gets the one. If the highest priority user is not the one who has just entered (Yasir), the system leaves the light value in room1 unmodified as it is already set to the preferred value of the highest-priority user (Maanta). And if the highest-priority user is the one who has just entered (Yasir), then application obtains his light preferred value. If the light value is less than 90% of his preferred value or greater than 110% of his preferred value, the light value is adjusted to his preferred value. The same process is repeated for any new users in room1.

As any of the users in room1 moves to other location, for example, in room2, the user presence context changes (user = "Maanta" location = "room2" enter_or_left = "enter"). In response to this event, the system discovers a light service in room2 and updates the user's binding with the light service in room1 to the light service in room2. If Maanta is the first user in the room2, the light is turned on and light value is adjusted to what it was when the light was last turned off in the room2. If there are other users already present in the room2, the light value in the room2 would already be adjusted to the preferred value of the highest-priority user. The light intensity measuring widget senses light intensity in room2 and sends it to the application. As a response to this event, the application gets the highest-priority user from the system.

The activity context monitor monitors the activity of the highest-priority user and generates an activity event based on what she is doing (e.g., reading, sleeping, watching tv, etc). In response to this event, the system, based on her activity, modifies the light value to her preferred value for that activity. For example, there are three users, Roya, Yasir and Maanta in room1 and the highest-priority user is Roya, and the light value is already set to Roya's general preferred value for the light. Now Roya is reading, and the activity context monitors this activity and sends activity event (activity = "reading" location = "room1"). As a response to this event, the system modifies the light value to Roya's preferred value for reading. If another user enters room1 and she has the highest-priority, the light value is modified to her preferred value if the light value is less than 90% or greater than 110% of her preferred value.

If any user is leaving room1/room2 and she is the highest-priority user, the light value in room1/room2 is adjusted to the user preferred value of next highest-priority user in room1/room2. For example, users, Maanta, Yasir and Roya are in room1 and the light value in the room1 is adjusted to Roya's preferred value as she is the highest-priority user. Now Roya

leaves room1, the user presence context widget detects her leaving and sends the user presence event (user = “Roya” location = “room1” enter_left = “left”). In response to this event, the system removes her from a list of users in room1 and gets the next highest-priority user, which is Yasir and then adjusts light value to his preferred value. As users keep leaving room1 and when the last user leaves the room1, the system turns the light off in room1. This scenario is illustrated in figure 4.1 below.

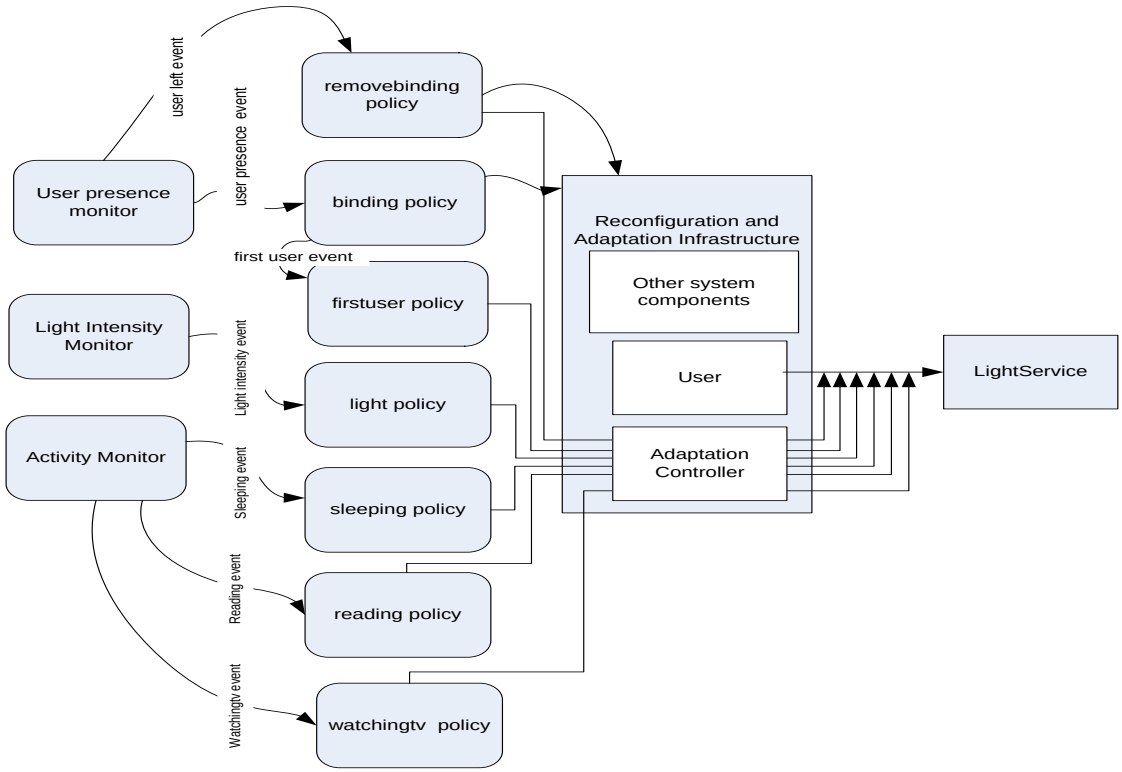

Figure 4.1: The home lighting example scenario

The source code that realizes this scenario within the PCRA is shown in figure 4.2.

```
lMAssEvent  lMEvent:  #("UserPresenceContext" "userpresence" "user" "location" "enter_or_left").
lMAssEvent  lMEvent:  #("LightIntensityContext" "lightintensity" "location" "lightSensed").
lMAssEvent lMEvent:   #("ActivityContext" "activityevent" "location" "activity").

//create an event type for the first user policy
event:= root/factory/event create: #("location").
root/event at: "firstuserevent" put:event.

// A binding policy
policy := root/factory/ecapolicy create.
policy event: root/event/userpresence.
policy condition: [:user :location :enter_or_left|(enter_or_left == "enter")].
policy action:[:user :location |
   theUser := users at: user.
   configurator createBinding: #(theUser location "LightService").
   root/event/firstuserevent create: #(location).].
root/policy at: "bindingpolicy" put: policy.
policy active: true.

//The first user policy
policy:=root/factory/ecapolicy create.
policy event: root/event/firstuserevent.
policy condition: [:location|
   IsFirstUser:=configurator IsFirstUser: #(location).
   (IsFirstUser)].
policy action:[:location  |
   adaptation performAdaptation:#("LightService" location "on" ).
   previousLightValue:= adaptation getRemoteFieldValue: #("LightService" location "getLightLevel").
   adaptation performAdaptation:#("LightService" location "adjust" previousLightValue).].
root/policy at: "firstuserpolicy" put: policy.
policy active: true.

// A policy to modify the light value to the user preferred value if the light sensed is lesser
//than 90% of the user preferred value or greater than 110% of the user preferred value.
policy := root/factory/ecapolicy create.
policy event: root/event/lightintensity.
policy condition: [:location :lightSensed |
   theUser:= configurator getHighestPriorityUser: #(location).
   preferredLightValue :=  theUser getPreferredValue: "LightService".
   IsUserSetEnabled:= theUser getExposed.
   (IsUserSetEnabled)&(lightSensed < ((90/100)*preferredLightValue)) |(lightSensed > ((110/100)*preferredLightValue))].
policy action:[:location :lightSensed |
   theUser:= configurator getHighestPriorityUser: #(location).
   preferredLightValue :=  theUser getPreferredValue: "LightService".
   adaptation performAdaptation:#("LightService" location "adjust" "userPreferredValue").].
root/policy at: "lightpolicy" put: policy.
policy active:true.

// policy to modify the behaviour of light service when the user is reading
policy := root/factory/ecapolicy create.
policy event: root/event/activityevent.
policy condition: [:location :activity |(activity=="reading")].
policy action: [:location :activity |
   theUser:= configurator getHighestPriorityUser: #(location).
   prefValReadActivity :=  theUser getPreferredValue: "LightService_reading".
   adaptation performAdaptation:#("LightService" location "adjust" prefValReadActivity).].
root/policy at: "readingpolicy" put: policy.
policy active: true.

// policy to modify the behaviour of light service of the room the user is sleeping
policy := root/factory/ecapolicy create.
policy event: root/event/activityevent.
policy condition: [:location :activity | (activity=="sleeping")].
policy action: [:location :activity |
   theUser:= configurator getHighestPriorityUser: #(location).
   prefValSleepActivity :=  theUser getPreferredValue: "LightService_sleeping".
   adaptation performAdaptation:#("LightService" location "adjust" prefValSleepActivity).].
root/policy at: "sleepingpolicy" put: policy.
policy active: true.

// policy to modify the behaviour of light service  when the user is watchingTV
policy := root/factory/ecapolicy create.
policy event: root/event/activityevent.
policy condition: [:location :activity |(activity=="watchingtv")].
policy action: [:location :activity |
   theUser:= configurator getHighestPriorityUser: #(location).
   prefValWatchTVActivity :=  theUser getPreferredValue: "LightService_watchingtv".
   adaptation performAdaptation:#("LightService" location "adjust" prefValWatchTVActivity).].
root/policy at: "watchingtvpolicy" put: policy.
policy active: true.

// user leaving policy
policy := root/factory/ecapolicy create.
policy event: root/event/userpresence.
policy condition: [ :user :location :enter_or_left|(enter_or_left=="left")].
policy action: [ :user :location :enter_or_left |
   configurator removeBinding: #(user location).
   adaptation performAdaptation:#("LightService" location "adjust" "userPreferredValue").].
root/policy at: "userleavingpolicy" put: policy.
policy active: true.
```

Figure 4.2: PCRA source code for the home lighting example scenario

4.1.1.2 The Light and Air-conditioning Scenario

This scenario has already been presented in chapter 3 to describe our approach to both contextual reconfiguration and contextual adaptation. In this section, we describe this scenario in detail along with source code.

When a user enters a particular room, the system turns on the light in the room and sets the light level to its last used level. The system also turns on the air-conditioning unit in the room and sets the air-conditioning setting to its last used value. Furthermore, the system turns on the light in adjoining rooms and set their light level to some low value. If the user stays in the room for some time (e.g., for a minute), the light level in that room is set to the user-preferred value for the light and air-conditioning setting is set to the user-preferred value for the air-conditioning, and the light in adjoining rooms are turned off. When the user leaves the room, the light and air-conditioning are turned off. We elaborate this application scenario in detail with an example.

The user, “Maanta” enters room1, the user presence context widget detects her presence and sends the user presence event (user = “Maanta” location = “room1” enter_or_left = "enter"). As a response to this event, the system creates the user instance for her, discovers a light service and air-conditioning service in room1 and a light service in an adjoining room (e.g. room2) and binds them all to the user instance. The system then turns on the light and air-conditioning in room1 and adjusts the light level and air-conditioning setting to their last used levels. The system also turns on the light in room2 and adjusts its level to some low value. At this point of time the system signals the timer to start counting time. If she stays longer than a minute in room1, the time event is fired. In response to this event, the system adjusts the light value and air-conditioning setting in the room1 to her preferred value for light and air-conditioning respectively and also turns off the light in room2. If she leaves room1, the user presence context widget detects this and sends the user presence event (user = “Maanta”, location = “room1” enter_or_left="left"). In response to this event, the system turns off the light and air-conditioning in room1, and removes bindings that she has with the light service and air-conditioning in room1 and the light service in room2. This scenario is illustrated in figure 4.3, while the source code for implementation of above scenario within the PCRA is given in figure 4.4.

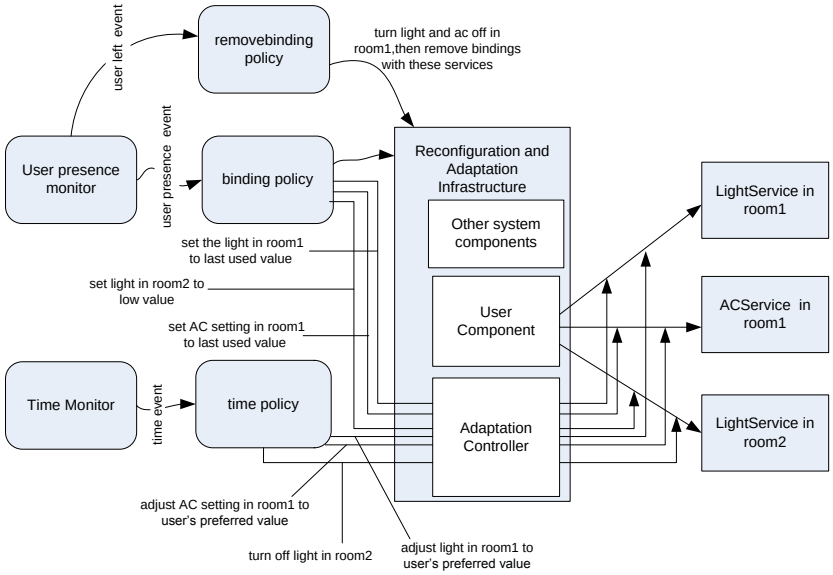

Figure 4.3: The light and air-conditioning scenario

```
lMAssEvent  lMEvent:  #("UserPresenceContext" "upevent" "user" "location" "enter_or_left").
lMAssEvent  lMEvent:  #("Timer" "timeEvent").

// A binding policy
policy := root/factory/ecapolicy create.
policy event: root/event/upevent.
policy condition: [:user :location :enter_or_left| (enter_or_left == "enter")].
policy action:[:user :location |
   theUser := users at: user.
   configurator createBinding: #(theUser location "LightService" "ACService").
   configurator createBinding: #(theUser "room2" "LightService" "otherLocation").
   adaptation performAdaptation:#("LightService" location "on").
   adaptation performAdaptation:#("ACService" location "on" ).
   previousLightValue:=adaptation getRemoteFieldValue:#("LightService" location "getLightLevel").
   previousAcSetting:=adaptation getRemoteFieldValue:#("ACService" location "getCurrentSetting").
   adaptation performAdaptation:#("LightService" "room2" "on" ).
   adaptation performAdaptation:#("LightService" location "adjust" previousLightValue).
   adaptation performAdaptation:#("ACService" location "adjust" previousAcSetting).
   adaptation performAdaptation:#("LightService" "room2" "adjust" 10).
   Timer sendEvent: 1.].
root/policy at: "bindingpolicy" put: policy.
policy active: true.

// policy that if the user stays a 1 min in e.g. room1, then switches off the light in  adjacent
// room(e.g. room2), adjusts the light  and the ac setting to the user preferred values in room1.
policy := root/factory/ecapolicy create.
policy event: root/event/timeEvent.
policy action: [
   location:= configurator getCurrentLocation.
   theUser:= configurator getHighestPriorityUser: location.
   otherLocation:= configurator getOtherLocation.
   preferredLightValue:= theUser getPreferredValue: "LightService".
   preferredAcSetting := theUser getPreferredValue: "ACService".
   adaptation performAdaptation:#("LightService" location "adjust" preferredLightValue ).
   adaptation performAdaptation:#("ACService" location "adjust" preferredAcSetting).
   adaptation performAdaptation:#("LightService" otherLocation "off").].
root/policy at: "timepolicy" put: policy.
policy active: true.

// user leaving policy
policy := root/factory/ecapolicy create.
policy event: root/event/upevent.
policy condition: [ :user :location :enter_or_left|(enter_or_left=="left")].
policy action: [ :user :location |
   Timer sendEvent: 0.
   adaptation performAdaptation:#("LightService" "room1" "off").
   adaptation performAdaptation:#("ACService" "room1" "off").
   adaptation performAdaptation:#("LightService" "room2" "off").
   configurator removeBinding: #(user location).].
root/policy at: "userleavingpolicy" put: policy.
policy active: true.
```

Figure 4.4: PCRA source code for the light and air-conditioning scenario

4.1.1.3 Extended Telephone, Light and Music Scenario

We have another application scenario which has been taken from Scooby [69] and slightly modified to add a little more complexity to it. Let us imagine that a user is sitting in a living room and listening to her favourite music in a CD player in dim light. Suddenly the phone starts ringing. A device is attached to the telephone, which generates the signals that indicate if the user has picked up the receiver to attend the call; the user has put down the receiver after the conversion is over or the user does not want to attend the call (the device either counts the number of beeps or notes the time). If the phone is not picked up within a certain number of beeps or within certain time period (e.g., 5 seconds), it is assumed that the user does not wish to attend the call. When the user picks up the receiver and attends the call, the system causes the light level to be returned to normal and for the volume in the CD player to be reduced. Once the user has finished talking and the receiver is placed back on the phone, the system modifies the light value to the previous level, and modifies the volume to what it was prior to the phone call. If the user does not attend the call, a voice message is recorded on the answering machine. We demonstrate this example scenario in more detail with an example.

The user, "Maanta" enters the living room, the user presence context widget detects her presence and sends the user presence event (user = "Maanta" location = "livingroom"). As a response to this event, the system creates the user instance for her, discovers a light service and CD player service in the living room and binds them to the user instance. The system then plays her favourite music in CD player, and the light value is set to some low value (dim light). When the phone starts ringing and she attends the call, the phone monitor event (attendingcall = "true") is fired. In response to this event, the system saves the current values of the light and music volume and then adjusts the light value to normal and the music volume to a very small value. When she finishes the conversion and places the receiver back on the phone, the phone monitor event (callfinished = "true") is fired. In response to this event the system modifies the light value and music volume to what they were before the phone call. If she does not pick the phone within a minute after the phone started ringing, the time event is fired. In response to this event, a voice message is recorded on the answering machine. When she leaves the living room, the user presence event (user = "none" location = "none") is fired. As a response to this event, the system turns off the light and the music, and removes bindings with these services.

We use this modified scenario in the qualitative part of evaluation chapter and show how modification and extensibility can be achieved both within the PCRA and within Scooby. This scenario is illustrated in figure 4.5 and the source code for implementation of above scenario within the PCRA is given in figure 4.6

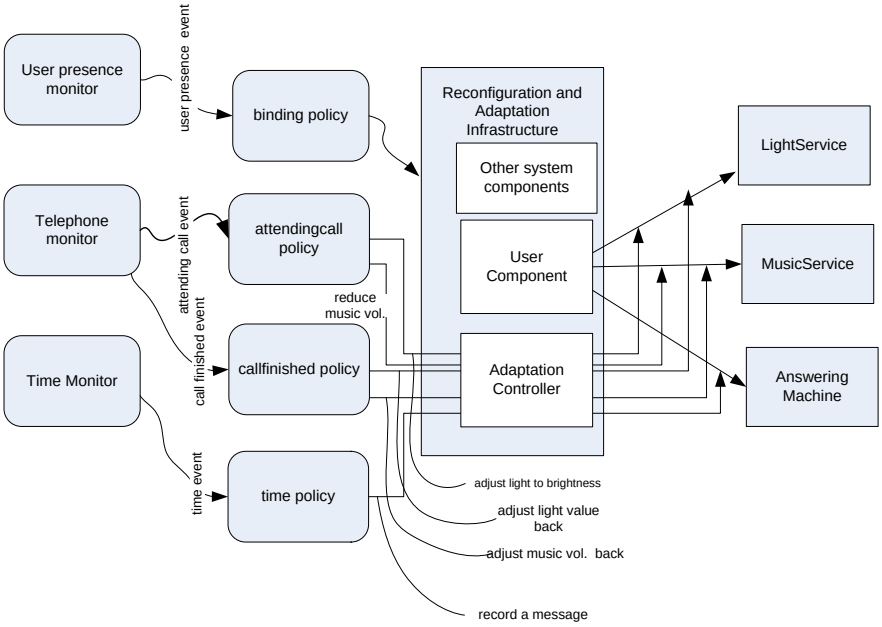

Figure 4.5: Extended telephone, light and music scenario

```
lMAssEvent  lMEvent:  #("UserPresenceContext1" "upevent" "user" "location").
lMAssEvent  lMEvent:  #("PhoneMonitor" "phoneEvent" "attendingcall" "callfinished").
lMAssEvent  lMEvent:  #("Timer" "timeEvent").

// The below line of code links the Timer (generates timeEvent)to the PhoneMonitor.
root/PhoneMonitor sendTimer: root/Timer.


// A binding policy to discover a light service, music service and answering machine service based on user's
//location and bind them to the user instance of the user in the location , and adjusts the light to dim
// value and plays music.
policy := root/factory/ecapolicy create.
policy event: root/event/upevent.
policy action:[:user :location |
   theUser := users at: user.
   configurator createBinding: #(theUser location "LightService" "MusicService" "AnsweringMachine").
   adaptation performAdaptation:#("LightService" location "adjust" 15).
   adaptation performAdaptation:#("MusicService" location "play").].
root/policy at: "bindingpolicy" put: policy.
policy active: true.

//A policy to change light value and volume of the music when the user attends a call.
policy := root/factory/ecapolicy create.
policy event: root/event/phoneEvent.
policy condition: [:attendingcall :callfinished | (attendingcall)].
policy action: [
   musicVol:= adaptation getRemoteFieldValue:#("MusicService" "livingroom" "getVolume").
   lightVal:= adaptation getRemoteFieldValue:#("LightService" "livingroom" "getLightLevel").

   // The variableSaver is the managed object that is used to save variables so that these variables
   // can later be obtained and used. The variables below (musicVol and lightVal) are saved in the
   //variableSaver object. These variables are later obtained from variableSaver object and used in
   //callfinshed policy below.

   variableSaver setLightValue: lightVal.
   variableSaver setMusicVolume: musicVol.
   adaptation performAdaptation:#("LightService" "livingroom" "adjust" 70).
   adaptation performAdaptation:#("MusicService" "livingroom" "setVolume" 0).].
root/policy at: "attendingcall" put: policy.
policy active: true.

//A policy to bring the light value and volume of the music back to what they were before user attended
//the call, when the user finishes talking and places receiver back on the phone.

policy := root/factory/ecapolicy create.
policy event: root/event/phoneEvent.
policy condition: [:attendingcall :callfinished |(callfinished)].
policy action: [
   lightVal:= variableSaver getLightValue.
   musicVol:= variableSaver getMusicVolume.
   adaptation performAdaptation: #("LightService" "livingroom" "adjust" lightVal).
   adaptation performAdaptation: #("MusicService" "livingroom" "setVolume" musicVol).].
root/policy at: "callfinshed" put: policy.
policy active: true.

//A policy to allow recording voice message if the phone is not attended within some time period(e.g., 5 sec)
// after the phone started ringing.
policy := root/factory/ecapolicy create.
policy event: root/event/timeEvent.
policy action: [
   adaptation performAdaptation: #("AnsweringMachine" "livingroom" "recordMessage" "message").].
root/policy at: "timepolicy" put: policy.
policy active: true.
```

Figure 4.6: PCRA source code for extended light, music and telephone scenario

4.1.2 Scooby Example Scenarios

In this section we introduce and describe high-level example scenarios that have been designed and implemented within the Scooby system [69]. As mentioned before, we have also taken the same scenarios and implemented them within the PCRA to make comparison. Scooby describes that each scenario is intended to build on the complexity of the preceding one; hence the last one is the most complex. In this section we reproduce the description of all five Scooby scenarios and then discuss how these scenarios are implemented within the PCRA along with the code.

4.1.2.1 Scenario 1: Simple printer service composition

The high-level description of the *simple printer service composition* is reproduced from the Scooby [69] and presented below.

"A user would like to print a PDF document on a printer. The required printer must be able to print with the following characteristics: ***print in colour****,* ***double-sided*** *and* ***print onA4-sized paper****."*

The requirement for this scenario involves discovering a printer service with above characteristics and document converter service, which can convert a given document in a PDF format, and binding them to the user instance. If the document is in a PDF format, it is directly sent to the printer service for printing. And if it is in other than a PDF format, it is sent to the converter service and then the converted PDF document returned by this service is sent to the printer service for printing. The realization of this scenario within the PCRA is illustrated in the following figure 4.7.

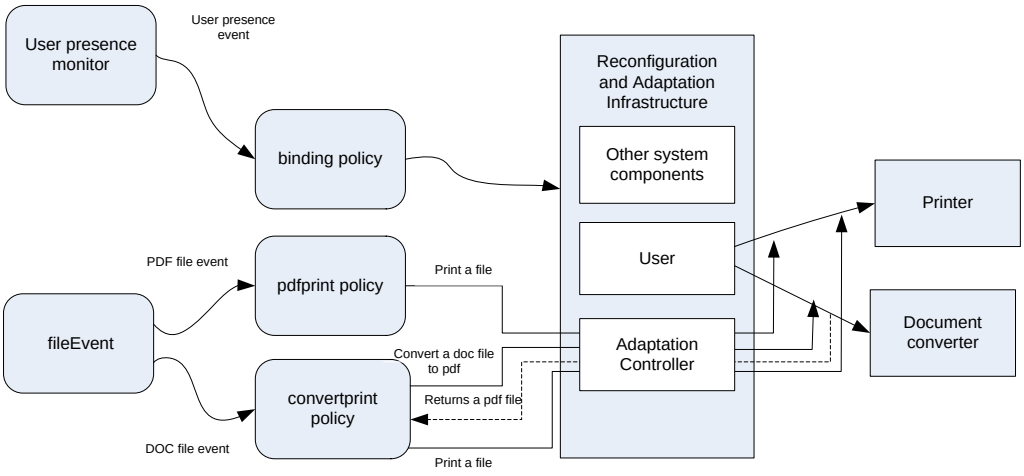

Figure 4.7: Printer and converter scenario

As can be seen in above diagram, the *binding* policy responds to user presence event (user, location) and discovers a printer service based on user's location (and other search attributes, colour, size, etc.) and a document converter service. There are two policies—*pdfprint* policy and *convertprint* policy that subscribe to the *fileEvent*. The *convertprint* policy checks if the document is a PDF file, it sends this file to the printer service to print. The *printconvert* policy checks if the document is not a PDF file (e.g., doc type), it sends this document to the converter service and the file is converted into PDF format and returned by this service, and then the policy sends the converted document to the printer service to print.

The source code that realizes this scenario within the PCRA is shown in figure 4.8, and Scooby source code (reproduced from Scooby thesis) for this scenario is shown in figure 4.9

```
lMAssEvent  lMEvent: #("UserPresenceContext1" "userpresence" "user" "location").
root/event at: "fileEvent" put:(root/factory/event create: #("file" "fileType")).

// A binding policy that discovers and binds to printer and converter service.
policy := root/factory/ecapolicy create.
policy event: root/event/userpresence.
policy action:[:user :location |
   theUser := users at: user.
   configurator createBinding: #(theUser location "Printer" "color" "A4" "serviceAttributes").
   configurator createBinding: #(theUser "Converter").
   root/event/fileEvent create: #("fileData" "pdf").
   root/event/fileEvent create: #("fileData" "doc").].
root/policy at: "bindingpolicy" put: policy.
policy active: true.

// A policy that sends a document to the printer to print if it is in a PDF format.
policy := root/factory/ecapolicy create.
policy event: root/event/fileEvent.
policy condition: [:file :fileType |(fileType=="pdf")].
policy action: [:file :fileType |
   adaptation performAdaptation: #("Printer" "sendMessage" file).].
root/policy at: "PDFprintpolicy" put: policy.
policy active: true.

// A policy that checks if the document is not in a PDF format, sends this to the converter
//service and then sends the converted PDF document to the printer service to print.
policy := root/factory/ecapolicy create.
policy event: root/event/fileEvent.
policy condition: [:file :fileType | (fileType!="pdf")].
policy action: [:file :fileType |
   converted:= adaptation getRemoteFieldValue: #("Converter" "convert" file).
   adaptation performAdaptation: #("Printer" "sendMessage" converted).].
root/policy at: "convertprintolicy" put: policy.
policy active: true.
```

Figure 4.8: PCRA source code for printer-converter scenario

```
service PDFPrinter decorates ptr: printer, converter: pdf2ps {
   bind ptr match { location : "5a22" & colour : "yes" & doublesided : "yes" &
   papersize : "a4"
}
{
    public void print( blob file ) {
    // print the file if it is a PDF
    if( file is "PDF" ) {
         ptr.print(file)
    }
    else {
         // convert the file
         file = converter.convert(file)
         ptr.print(file)
    }
 }
}
} when bindexception {
    reporterror( "error occurred" )
    terminate
}
```

Figure 4.9: Scooby source code for printer-converter scenario

4.1.2.2 Scenario 2: Follow Me Service

The high-level description of the *follow me service* is reproduced from the Scooby [69] and presented below.

"In this scenario, we propose a more complex set of interactions between services to provide the overall goal of directing information to a user, based on their proximity to a device. Let us imagine that: -

A user has configured a stock monitoring service to inform them when a stock price reaches a certain point. The service is configured to display the stock price on the closest available smart device to the user. This could be in the form of a message on a screen, printout, sms or email message, depending on the location of the user within the smart environment. It is assumed that the user may walk around within this environment and alter their location."

The requirement of this scenario involves discovering the closest rendering device service based on the location of the user and binding it to the user instance, and then sending the stock price information to the bound service of the closest available rendering device to the user. Figure 4.10 shows the realization of this scenario within the PCRA.

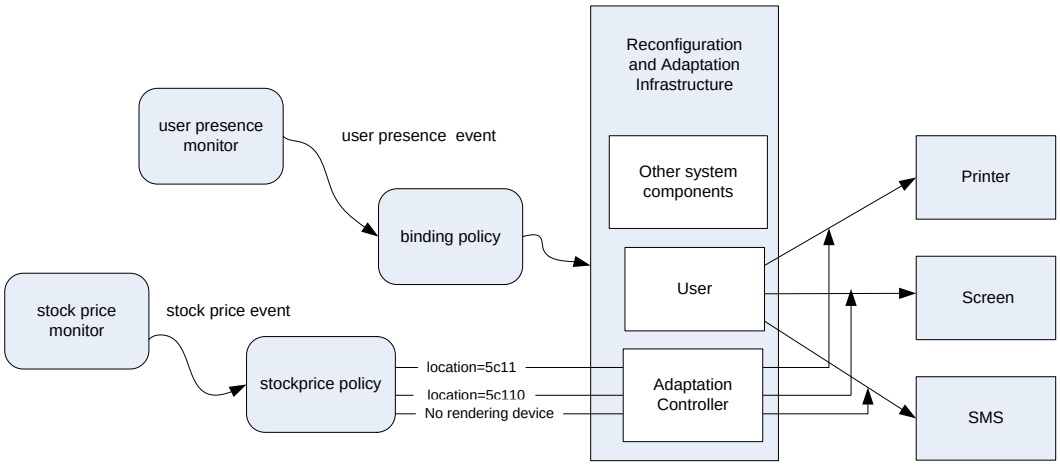

Figure: 4.10: Follow me service scenario

As can be seen in above diagram, there is a *binding* policy which responds to *user presence context (user, location)*. The user presence monitor detects the user presence in terms of the user's ID and location and generates a *user presence event*. In response to this event, this policy discovers the service for the rendering device based on the location of the user and binds it to the user instance. For example, a printer is located in room "5c11" and smart screen is fixed in room "5c10". If the user, "Maanta" enters room "5c11", the user presence context monitor detects her ID and location, and generates user *presence event (user = "Maanta" location ="5c11").* As a response to this event, the *binding* policy discovers the service for the printer in room "5c11" and binds it to the user instance. Similarly if she moves to room "5c10", the *binding* policy responds to *user presence event (user = "Maanta" location ="5c10")* and discovers the smart screen service in room "5c10" and binds it to the user instance. The stock price monitor monitors the stock price and when it reaches a certain point, it generates the *stock price event*. In response to this event, the *stockprice* policy sends the stock price to the device based on location of the user. If the current location of the user is room "5c10", the *stockprice* policy sends stock price to the smart screen. If the current location of the user is room "5c11", the *stockprice* policy sends the stock price to the printer. If the user is somewhere where there is no rendering device available, the *stockprice* policy sends an sms/email to the user.

The source code for implementation of this scenario within the PCRA is shown in figure 4.11 and Scooby source code (reproduced from Scooby thesis) for this scenario is shown in figure 4.12

```
lMAssEvent  lMEvent:  #("UserPresenceContext1" "userpresence" "user" "location").
lMAssEvent  lMEvent:  #("StockService" "stockpriceevent" "stockprice").

// A binding policy that discovers a rendering device based on location of the user
//and binds it to the user.
policy := root/factory/ecapolicy create.
policy event: root/event/userpresence.
policy action:[:user :location |
   theUser := users at: user.
   configurator createBinding: #(theUser location "bindingWithRenderingDevices").].
root/policy at: "bindingpolicy" put: policy.
policy active: true.

// A policy that responds to stock price event and sends stock price information
// to the rendering device closest to user location.
policy := root/factory/ecapolicy create.
policy event: root/event/stockpriceevent.
policy action: [:stockprice |
   location:=configurator getCurrentLocation.
   service:= configurator renderingService: #(location).
   adaptation performAdaptation: #(service location "sendMessage" stockprice).].
root/policy at: "stockpricepolicy" put: policy.
policy active: true.
```

Figure 4.11: PCRA source code for follow me scenario

```
service stockmonitor decorates cp : printer, cs: screen, sr : sms, em: email {

  // need to bind to an event which contains the current location of
  // the user. could have an ref id tag attached and associated service
  bind userlocation as event match { location: userloc }

  // we need to bind to relevant devices (printer, screen, sms)
  // depending on the location of the user
  bind cp match { location: printerloc }
  bind cs match { location: screenloc }
  bind stockservice as event match { price: stockprice }

 {
   int userpreference = 1
 }

{

// only fire when we get an event from the stock service
on notification( stockservice ) {

  // we need to call a relevant device depending on the location
  if( userloc == printerloc ) {
     cp.sendmessage( stockprice )
     }
  else {
    if( userloc == screenloc ) {
       cs.sendmessage( stockprice )
       }
     else {
       // depending on what the user wants, they will put code here to send
       // to email or sms
       if( userpreference ) { sr.sendmessage( stockprice ) }
          else {
            em.sendmessage( stockprice )
          }
     }
   }
  }
 }
} when bind exception() {
    reporterror("Error in binding.")
}
```

Figure 4.12: Scooby source code for follow me scenario

4.1.2.3 Scenario 3: The home coffee machine, fridge and cooker

The high-level description of the *home coffee machine, fridge and cooker* is reproduced from the Scooby [69] and presented below.

"A user initially informs the alarm clock of the time they wish to be woken. Depending on their policy, the alarm clock will inform the coffee machine a few minutes before the alarm is to go off, to tell it to start preparing the coffee, so that it is ready when the user wakes. In doing so, the coffee machine must request that there is enough milk from the fridge. The fridge will check the availability and will reply whether there is enough milk or not. If not, the fridge will inform the shopping list service by requesting it to add milk to the current list of items needing to be picked up next time a shopping run is scheduled. The coffee machine will then prepare the coffee and will then inform the cooker service that it needs to start preparing for breakfast. At this point the cooker is required to consult with the fridge to make sure that the components of the breakfast (in this case, bacon, eggs and bread) are available. Again, the fridge will check the availability of these items, and will add them to the shopping list if required. The cooker will then proceed in preparing the breakfast."

The realization of this scenario within the PCRA is illustrated in the following figure 4.13.

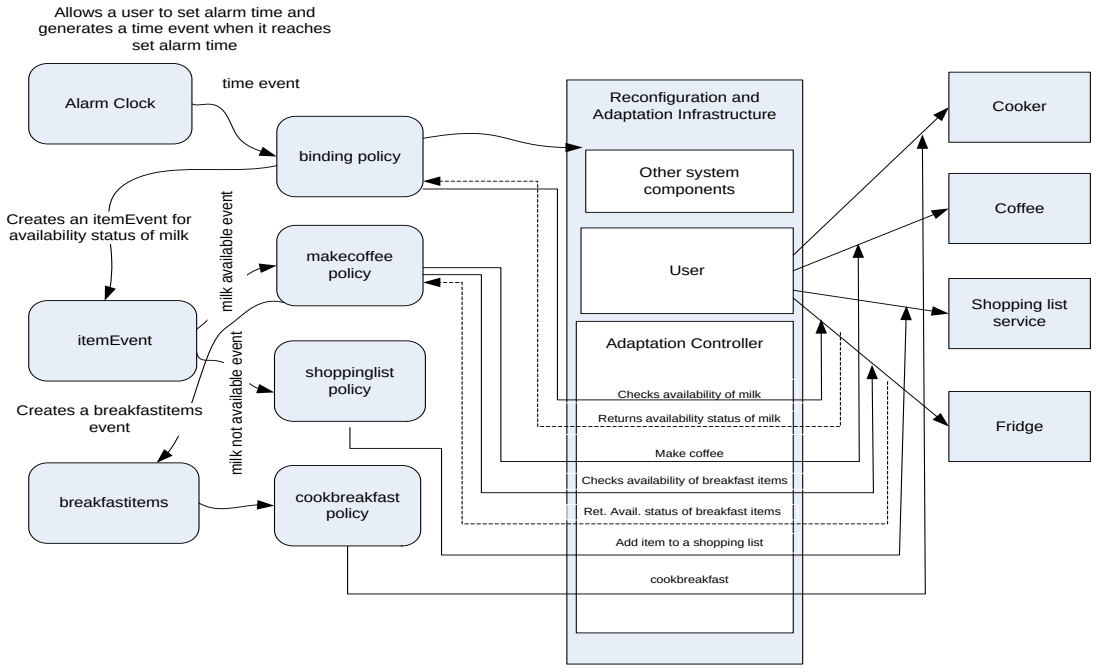

Figure 4.13: The home coffee machine, fridge and cooker

We have coded a managed object which is a GUI-based text clock. It also allows a user to set alarm time. When it reaches set alarm time, it generates a *time event*. The *binding* policy in this scenario as shown in figure 4.5 above responds to the *time event*. In response to *time event*, the *binding* policy discovers a coffee service, fridge service, cooker service and shopping list service and binds them all to the user instance. After the bindings have been established, the *binding* policy contacts the fridge service and checks the availability of the milk. It then creates an *itemEvent (item itemAvailability)* by filling in the name of the item and availability status of the item (item = "milk" itemAvailability = "true/false"). There are two policies that subscribe to this event: a *makecoffee* and *shoppinglist*. In response to this event, both the *makecoffee* and *shopinglist* policies get triggered and the one whose condition is true would be executed. If milk is available, the *makecoffee* policy would execute and issue a command to the coffee machine to start making coffee. If milk is not available, the *shoppinglist* policy would execute and add milk in the shopping list. After the *makecoffee* policy has issued the command to prepare coffee, it would contact the fridge service and get the availability status of each of breakfast items (e.g., eggs, bacon and bread). If the breakfast item is not available, the *shoppinglist* policy would be executed and add the breakfast item to the shopping list. After the *makecoffee* policy has the availability status of each of the breakfast items, it creates *breakfastitems (isBaconAvail isEggsAvail isBreadAvail)* event. In response to this event, the *cookbreakfast* policy gets triggered and checks if the breakfast items are available. If available, it issues a command to the cooker to start preparing breakfast.

The source code for implementation of this scenario within the PCRA is shown below in figure 4.14, and Scooby source code (reproduced from Scooby thesis) for this scenario is shown in figure 4.15.

```
lMAssEvent  lMEvent: #("TextClock" "timeevent").
root/event at: "itemEvent" put: (root/factory/event create: #("item" "itemAvailability")).
root/event at:"breakfastitems" put:(root/factory/event create:#("isBaconAvail" "isEggsAvail" "isBreadAvail")).

//A binding policy that discovers FirdgeService, CoffeeMachine, CookerService and ShoppingList in
//the kitchen and binds them to the user instance when the alarm clock starts ringing.
policy := root/factory/ecapolicy create.
policy event: root/event/timeevent.
policy action:[
   theUser := users at: "yasir".
   configurator createBinding: #(theUser "kitchen" "FridgeService" "ShoppingList"
   "CoffeeMachine""CookerService" ).
   isMilkAvail:= adaptation getRemoteFieldValue: #("FridgeService" "kitchen" "checkAvailability" "milk").
   root/event/itemEvent create: #("milk" isMilkAvail).].
root/policy at: "bindingpolicy" put: policy.
policy active: true.

// A policy to make coffee if milk is available in the fridge
policy := root/factory/ecapolicy create.
policy event: root/event/itemEvent.
policy condition: [:item :itemAvailability |(item=="milk")&(itemAvailability)].
policy action: [:item :itemAvailability |
   adaptation performAdaptation:#("CoffeeMachine" "kitchen" "prepare").
   isBaconAvail:= adaptation getRemoteFieldValue: #("FridgeService" "kitchen" "checkAvailability" "bacon").
   root/event/itemEvent create: #("bacon" isBaconAvail).
   isEggsAvail:= adaptation getRemoteFieldValue: #("FridgeService" "kitchen" "checkAvailability" "eggs").
   root/event/itemEvent create: #("eggs" isEggsAvail).
   isBreadAvail:= adaptation getRemoteFieldValue: #("FridgeService" "kitchen" "checkAvailability" "bread").
   root/event/itemEvent create: #("bread" isBreadAvail).
   root/event/breakfastitems create: #(isBaconAvail isEggsAvail isBreadAvail).].
root/policy at: "makecoffee" put: policy.
policy active: true.

// A policy to add a food item in the shopping list if not available in the fridge
policy := root/factory/ecapolicy create.
policy event: root/event/itemEvent.
policy condition: [:item :itemAvailability |(itemAvailability not)].
policy action: [:item :itemAvailability |
    adaptation performAdaptation:#("ShoppingList" "kitchen" "addToShoppingList" item).].
root/policy at: "shoppinglist" put: policy.
policy active: true.

// A policy to make breakfast if breakfast food items are available in the fridge
policy := root/factory/ecapolicy create.
policy event: root/event/breakfastitems.
policy condition: [:isBaconAvail :isEggsAvail :isBreadAvail |(isBaconAvail)&(isEggsAvail)&(isBreadAvail)].
policy action: [ :isBaconAvail :isEggsAvail :isBreadAvail |
    adaptation performAdaptation:#("CookerService" "kitchen" "cookBreakfast").].
root/policy at: "cookbreakfast" put: policy.
policy active: true.
```

Figure 4.14: PCRA source code for the home coffee machine, fridge and cooker

```
service coffeeMachine decorates fs: fridge, sls: shoppinglist, cs: cooker {
   bind fs match { location: "kitchen" }
   bind cs match { location: "kitchen" }
   {
      location: "kitchen"
   }

   {
     public void prepare() {
       if(fs.checkAvailability( "milk" ) ) {
         display("Preparing coffee...")
         cs.cookBreakfast()
       } else {
          display("There is not enough milk in the fridge, so need to order some!")
          sls.order("milk")
          }
    }
   }
} when bindexception {
   reporterror( "Bind exception occurred..." )
  }

service cooker decorates fs: fridge, sls: shoppinglist {
  bind fs match { location: "kitchen" }
  {
    location: "kitchen",
    cooker_type: "electric"
  }

  {
    public void cookBreakfast() {
      if(fs.checkAvailability( "bacon" ) ) {
        if(fs.checkAvailability( "eggs" ) ) {
          if(fs.checkAvailability( "bread" ) ) {
          display("Making breakfast...")
        }
        else { sls.order("bread") }
        } else { sls.order("eggs") }
      } else { sls.order("bacon") }
    }
 }
} when bindexception {
    reporterror( "Unable to bind..." )
}
```

Figure 4.15: Scooby source code for the home coffee machine, fridge and cooker

4.1.2.4 Scenario 4: The home environment

The high-level description of this scenario is reproduced from Scooby [69] and presented below.

"The user is in their car, returning home from work. A PDA is present within the car and is connected to a GPS system allowing a calculation of the time remaining before the user reaches home. Ten minutes before arriving, the PDA signals the home, telling it of the user's imminent arrival. Upon doing this, devices in the home activate. The heating device turns on so that the house is warm and hot water is available. The lights are turned on in the garage and entrance hall of the house. The curtains are automatically drawn and the coffee machine starts preparing some fresh coffee. As soon as the user reaches home and enters the house, the garage lights turn off as they leave the garage, motion detectors track the user's movements in the house, turn on the living room lights and start playing their chosen piece of music in the CD player."

This scenario involves discovering various services at home, binding them and performing various actions on these services in order to realize this scenario when the arrival time of the user at home falls below 10 minutes. The figure 4.16 below illustrates the realization of this scenario within the PCRA.

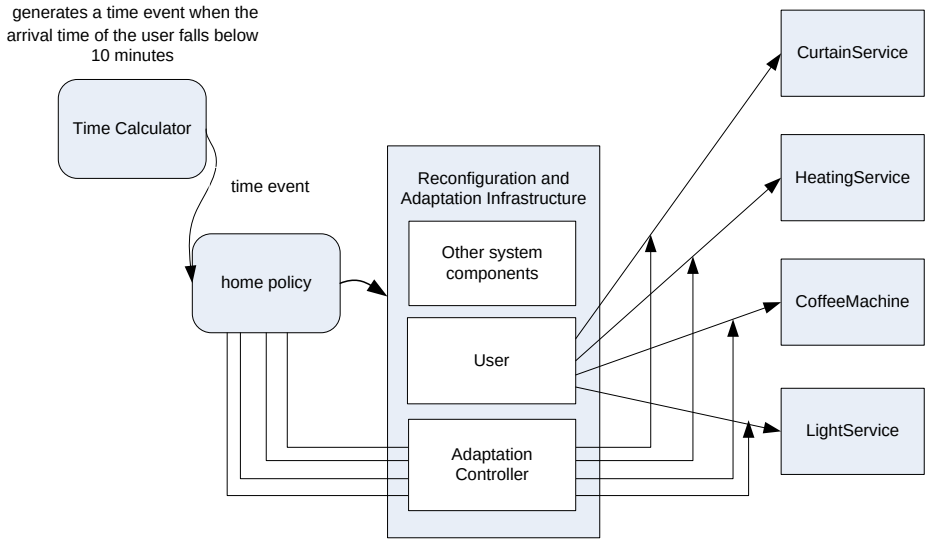

Figure 4.16: Home environment scenario

We have simulated this part of the scenario "*Ten minutes before arriving, the PDA signals the home, telling it of the user's imminent arrival"* by coding a managed object. This managed object is a GUI-based time calculator which takes the arrival time of the user as an input and generates a *time event* when the arrival time falls below 10 minutes. In response to *time event*, the *home* policy is triggered and discovers a coffee service, heating service, curtain service and light service and binding them all to the user instance. Once bindings have been created, the *home* policy performs invocations on the bound services to realize the behaviour of this scenario. The source code for realization of this scenario within the PCRA is shown in the figure 4.17, while the Scooby source code (reproduced from Scooby thesis) for this scenario is shown in figure 4.18.

```
lMAssEvent  lMEvent:  #("TimeCalculator" "time" ).
policy := root/factory/ecapolicy create.
policy event: root/event/time.
policy action:[
   user := users at: "Yasir".
   configurator createBinding:#(user "frontroom" "CurtainService" "LightService" "noUserLocation").
   configurator createBinding:#(user "bedroom" "CoffeeMachine" "noUserLocation").
   configurator createBinding: #(user "45Lansdowne" "HeatingService" "noUserLocation").
   adaptation performAdaptation:#("LightService" "frontroom" "on").
   adaptation performAdaptation:#("HeatingService" "45Lansdowne" "on").
   adaptation performAdaptation:#("CurtainService" "frontroom" "closeCurtains").
   adaptation performAdaptation:#("CoffeeMachine" "bedroom" "prepare").].
root/policy at: "homepolicy" put: policy.
policy active: true.
```

Figure 4.17: PCRA source code for home environment scenario

```
service myPolicy decorates lighting: lightingcontroller, curtains:
curtaincontroller, heating: heatingcontroller, coffee: coffeemachine {

  bind eta as event match { arrivaltime: "10" }
  bind curtains match { location: "frontroom" }
  bind coffee match { location: "bedroom" }

  {

  // this service is kicked off by an event, so put in handler
  on notification( eta ) {
    lighting.turnLightsOn()
    curtains.closeCurtains()
    heating.turnOnHeating()
    coffee.prepare()
  }
 }
} when bindexception {
    reporterror("Bind exception occurred...")
}
```

Figure 4.18: Scooby source code for home environment scenario

4.1.2.5 Scenario 5: The music & telephone scenario

The high-level description of this scenario is reproduced from the Scooby [69] and presented below.

"The user is sitting in a dimly lit living room, listening to music. Suddenly the phone starts ringing. Connected to the phone is a device that detects an incoming call. This automatically causes the lighting levels to be returned to normal (if previously dimmed) and for the volume in the CD player to be reduced. The user picks up the phone and begins to talk. Once the conversation has finished, and the receiver is placed back on the phone, the lighting is returned to the previous levels, and the volume returns to what it was before the phone call."

In order to simulate that whether the phone is ringing and being attended and the conversion has finished (receiver is placed back on the phone), we have implemented a managed object which is a GUI-based component and acts as a phone monitor. This component generates a *phone event* when a button for a particular action is pressed (e.g., by pressing a "*Call Finished*" button). This scenario involves a *binding* policy which responds to the *user presence context event* and discovers the music service and light service and binds them to the user instance. The other two policies involved are *attendingcall* and *callfinsished* policies and both of these respond *to phone event* context. The *attendingcall* policy modifies the light value and the volume of the music when the user is attending the call, while the *callfinished* policy brings the light value and the volume of the music back to what they were before the user attended the call, once the user has finished the conversion and placed back the receiver on the phone. Figure 4.19 below illustrates the realization of this scenario within the PCRA.

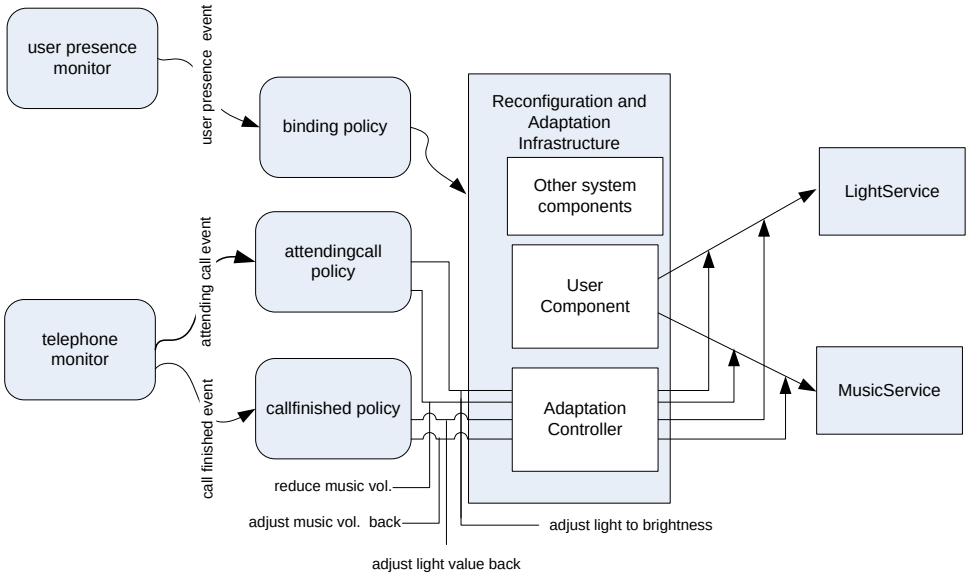

Figure 4.19: The music & telephone scenario

The source code for realization of this scenario within the PCRA is shown in the figure 4.20, while the Scooby source code (reproduced from Scooby thesis) for this scenario is shown in figure 4.21.

```
lMAssEvent  lMEvent:  #("UserPresenceContext1" "upevent" "user" "location").
lMAssEvent  lMEvent:  #("PhoneMonitor1" "phoneEvent" "attendingcall" "callfinished").

// A binding policy that discovers the light service and the music service and bind
// them to the user instance

policy := root/factory/ecapolicy create.
policy event: root/event/upevent.
policy action:[:user :location |
   theUser := users at: user.
   configurator createBinding: #(theUser location "LightService" "MusicService").
   adaptation performAdaptation:#("LightService" location "adjust" 15).
   adaptation performAdaptation:#("MusicService" location "play").].
root/policy at: "bindingpolicy" put: policy.
policy active: true.

//A policy to change the light value and volume of the music when the user attends a call
policy := root/factory/ecapolicy create.
policy event: root/event/phoneEvent.
policy condition: [:attendingcall :callfinished |(attendingcall) ].
policy action: [
   musicVol:= adaptation getRemoteFieldValue:#("MusicService" "livingroom" "getVolume").
   lightVal:= adaptation getRemoteFieldValue:#("LightService" "livingroom" "getLightLevel").
   variableSaver setLightValue: lightVal.
   variableSaver setMusicVolume: musicVol.
   adaptation performAdaptation:#("LightService" "livingroom" "adjust" 70).
   adaptation performAdaptation:#("MusicService" "livingroom" "setVolume" 0).].
root/policy at: "attendingcall" put: policy.
policy active: true.

// A policy to bring the light value and the volume of the music back to what they were before
//the user attended the call, when the user has finished the conversion and placed back the
//receiver on the phone

policy := root/factory/ecapolicy create.
policy event: root/event/phoneEvent.
policy condition: [:attendingcall :callfinished |(callfinished)].
policy action: [
   lightVal:= variableSaver getLightValue.
   musicVol:= variableSaver getMusicVolume.
   adaptation performAdaptation: #("LightService" "livingroom" "adjust" lightVal).
   adaptation performAdaptation: #("MusicService" "livingroom" "setVolume" musicVol).].
root/policy at: "callfinshed" put: policy.
policy active: true.
```

Figure 4.20: PCRA source code for the music & telephone scenario

```
service telephone decorates cd: cdplayer, lighting: lightingcontroller {
       bind cd match { location: "5a22" }
       bind lighting match { location: "frontroom" }
       bind noPhone as event match { status: "idle" }
       bind phoneInUse as event match { status: "ringing" }
     {
            location:"5a22"
     }
    {
            int lightingLevel
            int volumeLevel
    }
   {
       on notification( phoneInUse ) {
            volumeLevel = cd.getVolume()
            lightingLevel = lighting.getBrightnessLevel()
            cd.setVolume( 0 )
            lighting.turnLightsOn()
            }
       on notification( noPhone ) {
            lighting.setBrightnessLevel( lightingLevel )
            cd.setVolume( volumeLevel )
            }
   }
} when bindexception {
         reporterror( "Binding exception occurred..." )
}
```

Figure 4.21: Scooby source code for the music & telephone scenario

4.2 Scooby Description Outline

In the previous section, we provided high-level description of all Scooby scenarios and the source code for realization of these scenarios on PCRA and on Scooby. In this section we use the Scooby code for the music & telephone scenario (figure 4.21) and take a look at how a Scooby composite service is constructed.

The Scooby service has a distinct structure as there are several sections used to describe specific areas of functionality. We divide the composite service code for the music & telephone scenario into sections that describe the specific areas of functionality of a Scooby service, and briefly take a look at each of these sections.

The first section of the Scooby service is a binding variable declaration block where binding variables are defined and used globally throughout the service. The binding variable declaration block of above scenario is shown in figure 4.22

```
// binding variable declaration block

bind cd match { location: "5a22" }
bind lighting match { location: "frontroom" }
bind noPhone as event match { status: "idle" }
bind phoneInUse as event match { status: "ringing" }
```

Figure 4.22: Binding variable declaration block

One of the core concepts in the Scooby is the use of binding variables which provide high-level means to discover and bind services based on some search criteria. As can be noted in figure 4.22, there are four binding variables involved in the music & light example scenario: two service binding variables and two event binding variables. The service binding variable provides a link between a service and remote service and the remote service is accessed through its corresponding binding variable. The event binding variable is used in the service to capture an event. The high-level language constructs such as binding variables and others have a direct mapping to the underlying processes present within the Scooby middleware.

What follows the binding variable declaration block is a service characteristics block, which contains characteristics that will be associated with the service when it sends its service description and this description is used in the process of service discovery. The service characteristics block is shown in figure 4.23.

```
// service characteristics block

{

     location:"5a22"

}
```

Figure 4.23: Service characteristics declaration block

The characteristics that are defined in this section can be static or dynamic. If the characteristics of the service never change throughout its lifetime, these would be defined statically, and those which change throughout the service's lifetime would be defined dynamically.

What follows the service characteristics block in the example scenario is a variables declaration block. This block contains a list of variables, and these variables are global variables and are used throughout the service. This block is shown in figure 4.24.

```
// variables declaration block
{
     int lightingLevel
     int volumeLevel
}
```

Figure 4.24: Variables declaration block

After these declaration blocks is a main code section and this is used to define any methods, notification handlers or binding handlers within the Scooby service. The music &

telephone scenario (figure 4.21) does not have any methods. However, methods can be defined either private or public, and are used to set their visibility as local or external. Event handlers can be defined in the main section to perform event processing upon the occurrence of an event. The music & telephone scenario includes two notification handlers as shown in figure 4.25.

```
on notification( phoneInUse ) {
       volumeLevel = cd.getVolume()
       lightingLevel = lighting.getBrightnessLevel()
       cd.setVolume( 0 )
       lighting.turnLightsOn()
      }

on notification( noPhone ) {
       lighting.setBrightnessLevel( lightingLevel )
       cd.setVolume( volumeLevel )
      }
```

Figure 4.25: Notification handlers

The final section within the Scooby service is an exception-handling block. This block provides a place to include the code about what to be done if a binding reference becomes unsatisfied during the life cycle of the service. This block is shown in figure 4.26.

```
when bindexception {
       reporterror( "Binding exception occurred..."
)
```

Figure 4.26: Binding exception code block

When the binding ultimately fails, this binding exception code block is executed which allows the developer to specify what action to perform. For example, this may include to terminate the service, start the discovery process again or to put in additional code to change the search criteria or something.

4.3 Discussion and Summary

In this chapter we have provided high-level description of several hypothetical example scenarios that we have implemented along with source code for each scenario. Additionally, we also discussed briefly implementation of all Scooby scenarios within PCRA, and these scenarios are used in evaluation chapter for further analysis and comparison with Scooby. As can be observed from these scenarios, the scenarios involve both adaptive context-aware features of context-awareness (contextual reconfiguration and contextual adaptation), and their

implementation within PCRA indicates that PCRA combines and provides the support for both, thereby supporting the first argument of this thesis (i.e., providing a broader scope of adaptation by combining and providing the support for both contextual reconfiguration and contextual adaptation).

It is stated in the Scooby thesis that each scenario implemented within the Scooby is intended to build on the complexity of the preceding one and vary in scope and complexity from a simple printer composition to more complex ones. As the last scenario is the music & telephone scenario, this is the complex of all the scenarios. We have extended this scenario to add more complexity to it and presented in section 4.1.3 as one of PCRA scenarios. Implementation of all scenarios (PCRA and Scooby) provides enough evidence to argue that the PCRA can handle diverse and complex scenarios.

In this chapter we also looked at how a Scooby composite service was constructed and discussed various sections used within the Scooby service. Each of the sections is used to describe the specific areas of functionality of a Scooby service (i.e., binding variable block to define binding variables, service characteristics block to define service characteristics and the main code section to define any methods, notification handlers and binding handlers).

Unlike a Scooby service in which various sections are used to describe the specific areas of functionality, as can be observed from the source code for various example scenarios implemented within PCRA, there are no dedicated sections. Polices can be described in any order as these are independent units of execution and are to be executed in response to context. However, the other code related to creating event templates and associating them with context monitors have to be defined before the policies that respond to these contexts.

Chapter 5

Prototype Implementation

In the previous chapter, we provided a high-level description of several hypothetical example scenarios which we have implemented within our proposed system, PCRA, along with the source code. In this chapter we discuss the implementation of the home lighting example scenario, which was presented along with source code in the previous chapter (section 4.1.1.1), to demonstrate our contributions.

As discussed in chapter 3, the overall architecture of PCRA is comprised of three elements: the Ponder2 system, our reconfiguration and adaptation infrastructure, and Java RMI. We have designed and implemented our reconfiguration and adaptation infrastructure within PCRA as a collection of several Java managed objects (reconfiguration manager, adaptation controller, user component). The implementation of the reconfiguration and adaptation infrastructure also includes other system components which are implemented as conventional Java objects (virtual stub, virtual stub cache manager). The development of adaptive context-aware applications within PCRA involves the coding of binding and adaptation policies in Ponder2. In response to these binding and adaptation policies, the Ponder2 system directly interacts with the managed objects of the reconfiguration and adaptation infrastructure, and the managed objects interact with non-managed objects of the infrastructure to realize the overall goal of providing policy-based context-aware reconfiguration and policy-based context-aware adaptation within PCRA.

We have also designed and implemented a GUI-based system utility that provides various features that allow performing various tasks that are required before any example scenario is executed on PCRA. In this chapter, we also discuss PCRA's features and implementation details. Realizing the adaptive context-aware example scenarios, we have implemented required responding to various contexts, such as a user presence, user activity, time, phone context, stock price context, etc. In order to provide contextual information dynamically to the applications, we have implemented various managed objects which act as simulated GUI-based context monitors. For example, the user presence context monitor is used to create and send user presence context (user, location). Similarly, the phone monitor is used to

simulate whether the phone is being attended or the user has stopped conversation and placed the receiver back on the phone. This information is sent to the system by pressing a corresponding button. For example, contextual information that a user is attending call is sent to the application by pressing an "Attending Phone" button, and information that the user has finished talking is sent to the application by pressing a "Call Finished" button. This chapter also briefly discusses the implementation of some of these context monitors.

5.1 System Utility

In this section, we discuss the features and implementation details of our GUI-based system utility. This utility offers the following features:

- It allows creating and adding user instances to and removing from the PCRA environment without stopping or interrupting the system.
- It enables end users to dynamically customize their preferences for various services in the environment (e.g., the light service, the air-conditioning service, etc.).
- It allows loading and executing example scenarios through the GUI.

Figure 5.1 depicts the GUI of the system utility.

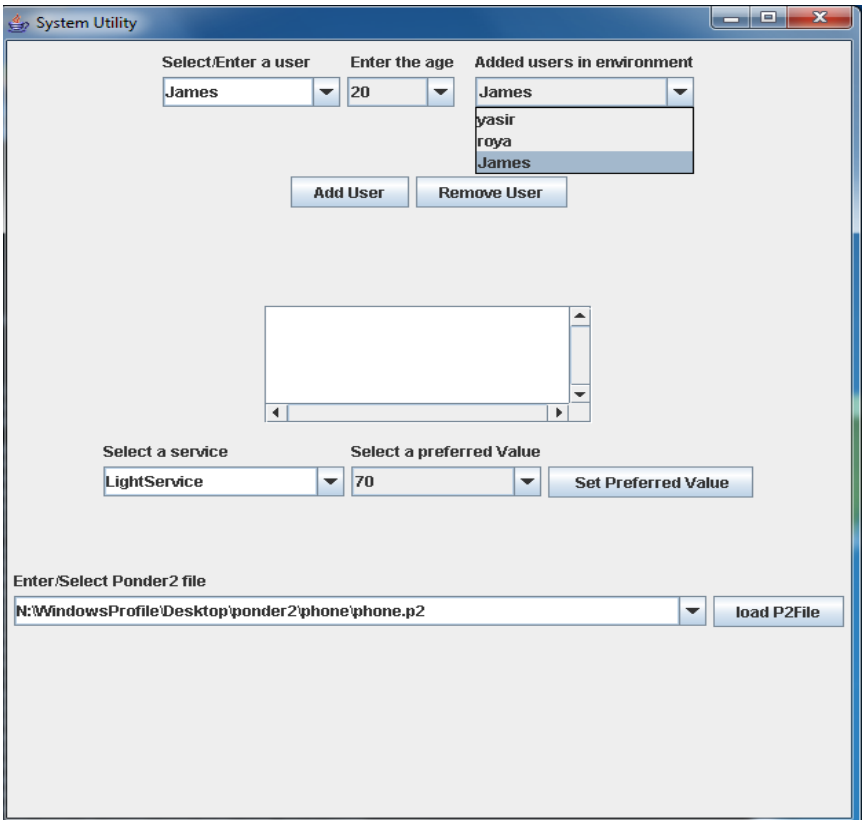

Figure 5.1: GUI of the system utility

The design of adaptive context-aware applications within PCRA involves the user of the environment, in which the system discovers remote service(s) based on her location (or her location and other search criteria) and binds them to her user instance (an instance of user component). As discussed in chapter 3, the user component, one of the architectural components of the reconfiguration and adaptation infrastructure, is used to model the users of the environment. This requires having user instances created and added to the system before running the example scenarios. The system utility allows this to be done dynamically, and also allows dynamic removal of user instances from the system through its GUI. As a part of the functionality of the applications, the user instance is obtained from a list of previously saved user instances based on her ID, and remote service(s) are discovered and bound to this user instance. Among other functionality, this component also saves the user preferences for various services and includes code through which the user can set or customize her preferences dynamically through the system utility GUI. In order to execute example scenarios, and to dynamically modify them if required, the system utility also allows loading ponder2 files into the system dynamically through its GUI.

To develop this utility we have coded a system utility GUI component in Java (SystemUtility.java) which is a ponder2 managed object, and a ponder2 file called *system_utility.p2*. System_Utility.java displays the GUI shown in figure 5.1. By pressing a button on the GUI, an event is generated and the corresponding policy in the *system_utility.p2* file is executed, realizing the required behaviour. For example, by choosing a user from the "Enter/Select a user" combo box and then pressing "Add User" button, an *addRemUser* event is generated causing an *adduser* policy to execute, which creates a user instance for the user and adds her to the PCRA environment. In addition to ponder 2 code (various event templates and policy specifications, etc.) related to the functionality of the system utility, the *system_utility.p2* file also includes code for loading the reconfiguration and adaptation infrastructure in the Ponder2 environment. We divide the source code of the *system_utility.p2* into two parts: the first includes code to load our reconfiguration and adaptation infrastructure and to create various event types (figure 5.2), and the second includes all policies (figure 5.3).

```
(1)
factoryObject =:root load: "LoadMonitor_AssociateEvent".
root/factory at: "lMAssEvent"  put: factoryObject.
root at: "lMAssEvent" put: root/factory/lMAssEvent create.

(2)
factoryObject := root load: "Configurator".
root/factory at: "configurator" put:factoryObject.
root at: "configurator" put: root/factory/configurator create.

(3)
factoryObject := root load: "AdaptationController".
root/factory at: "adaptationcontroller" put:factoryObject.
root at: "adaptation" put:(root/factory/adaptationcontroller create: root/configurator).

(4)
factoryObject:= root load: "User".
root/factory at: "user" put:factoryObject.
root at: "users" put: root/factory/domain create.

(5)
event := root/factory/event create: #("user" "age" "op").
root/event at: "addRemUser" put: event.

(6)
factoryObject := root load: "System_Utility".
root/factory at: "systemutility" put:factoryObject.
root at: "systemutility" put: (root/factory/systemutility event: root/event/addRemUser).

(7)
event := root/factory/event create: #("user" "service" "value").
root/event at: "setPreferredValue" put: event.
root/systemutility setPreferredValueEvent: root/event/setPreferredValue.

(8)
factory:= root load: "VariableSaving".
root/factory at: "variableSaver" put:factory.
root at: "variableSaver" put: root/factory/variableSaver create.
```

Figure 5.2: First part of system_utility.p2 code

We briefly explain what each part in the code shown in figure 5.2 does.

1. Load and instantiate the *LoadMonitor_AssociateEvent*, which is a managed object we have implemented. This component generates the code behind the scenes that loads the context monitor, creates an event type and assigns it to the context monitor. We discuss this later. In Ponder2, the word *root* denotes top-most domain. The *factoryObject =: root load: "LoadMonitor_AssociateEvent"* loads *LoadMonitor_AssociateEvent* managed object and returns its corresponding factory object (see 2.6.2.3 for discussion on factory object concept in Ponder2). In the *root/factory*, the *factory* is a domain, which was created besides other domains when the Ponder2 system was booted (read the section 5.3 for further details). The *at:put:* is the domain keyword message that puts an object in the domain. The *root/factory at: "lMAssEvent" put: factoryObject* puts *factoryObject* into *root/factory* domain, giving the name *lMAssEvent.* This factory instance (*root/factory/IMAssEvent*) is used to create the instances of *LoadMonitor_AssociateEvent* managed object. The *root at: "lMAssEvent" put: root/factory/lMAssEvent create* creates an instance of *LoadMonitor_AssociateEvent* managed object and puts it into *root* domain, giving the

name *lMAssEvent*. Note that *root/lMAssEvent* is an instance of *LoadMonitor_AssociateEvent* manage object, which can be used to send messages to it, while *root/factory/lMAssEvent* is the factory object of *LoadMonitor_AssociateEvent* manage object, which is used to create its instances.

2. Load the reconfiguration manager code, which is one of the system components of the reconfiguration and adaptation infrastructure, and create its instance. The functionality of the reconfiguration manager is accessed through its various messages (e.g., createBinding (args)).

3. Load the adaptation controller code, another system component of the reconfiguration and adaptation infrastructure, and create its instance by passing it an instance of the reconfiguration manager as an argument. The adaptation controller has the reconfiguration manager as one of its instance variables, which is initialized when an instance of the adaptation controller is created.

4. Load user component code and create a domain instance called *"users"*. Domain is one of the pre-defined types of managed objects in Ponder2, and when Ponder2 environment is loaded all pre-defined Ponder2 managed objects are loaded. It can be noted that the user component is loaded only and its instance(s) are not created yet. However, as discussed before, user instances will be created and added to the PCRA environment through the GUI-based system utility.

5. Create an *addRemUser* event type for the system utility to send. This event type has three attributes (user, age, op). The op is an operation type that can be either "add" or "remove".

6. Load the system utility code and give it the *addRemUser* event template. As discussed before, System_Utility.java program displays the GUI and generates an *addRemUser* event when either the "Add User" or "Remove User" button is pressed.

7. Create *a setPreferredValue* event type and give it to system utility to send. This event type has three attributes (user, service, value).

8. Load the VariableSaving class code and create its instance, variableSaver. This is a simple managed object we have created, which has been used in both versions of music and telephone scenario (figure 4.6 and 4.20) to save a light value and music volume so that these can later be obtained and used.

The second part of the source code of system_*utility.p2* is shown below in figure 5.3.

```
(8)
policy := root/factory/ecapolicy create.
policy event: root/event/addRemUser.
policy condition: [:user :age :op | (op=="add")].
policy action: [:user :age :op |
     domainUser := root/factory/user create: #(user age).
     root/users at: user put: domainUser.].
root/policy at: "adduser" put: policy.
policy active: true.

(9)
policy := root/factory/ecapolicy create.
policy event: root/event/addRemUser.
policy condition: [:user :age :op |(op=="remove")].
policy action: [:user :age :op |
     root/users remove: user.].
root/policy at: "removeuser" put: policy.
policy active: true.

(10)
policy := root/factory/ecapolicy create.
policy event: root/event/setPreferredValue.
policy action: [:user :service :value |
     theUser := users at: user.
     theUser setPreferredValue: #(service value).].
root/policy at: "setpreferredvalue" put: policy.
policy active: true.
```

Figure 5.3: The second part of the system_utility.p2 source code

The second part of the *system_utility.p2* source code (figure 5.3) includes specification of three policies involved in the system utility application. We now briefly explain what behaviour each policy realizes.

9. The *adduser* policy subscribes to the *addRemUser* event type. When the user and her age are selected from the system utility GUI and the "Add User*"* button is pressed, the GUI creates and sends an *addRemUser* event. For example, if the user "James" is selected, his age is set (e.g., 20), and the "Add User" button is pressed, the *addRemUser(user = "James" age = "20" op = "add")* event will be created and sent. In response to this event, both *adduser* and *removeuser* policies will be triggered as both subscribe to this event and the one whose condition is true will be executed. In this case the *adduser* policy will be executed as its condition is true. As a result, this policy creates a user instance and then puts it into the domain called *"users"* (the domain *"users"* was created previously, refer to (4) in figure 5.2).
10. The *removeuser* policy removes the user from the domain "*user"*. When the user from an existing list of users is selected and the "Remove User" button is pressed, the *addRemUser* event is fired with its operation variable being set to "remove". In response to this event, the *removeuser* policy gets executed and removes the selected user from the domain *"users"*.

11. The *setpreferredvalue* policy subscribes to the *setPreferredValue* event. When a user from the list of existing users, a service and the preferred value for this service is selected, and the "Set Preferred Value" button is pressed, the *setPreferredValue* event is fired. In response to this event, the *setpreferredvalue* policy gets executed and performs the following actions:

    - It retrieves the user instance for the selected user
    - The *setPreferredValue: #(service value)* message is mapped into *void setPreferredValue(P2Object args)* method call, which is one of the methods of user component. This one argument (i.e., args of P2Object) will hold all the attributes of the message (i.e., a service name and a preferred value for this service). This argument is converted into a P2Object array and then all attributes of the message are retrieved from this array. The execution of this method call results in setting up the preferred value for the service in the user instance for the selected user.

5.2 Simulated Context Monitors

As discussed above, the application scenarios we have implemented involve responding to various contexts to realize the required behaviour. We have implemented various managed objects which simulate different context monitors and provide contextual information dynamically to the applications. As discussed in chapter 2, Ponder2 has an event model in which an event can be used as a means of passing contextual information to interested parties (e.g., policies). In order for a context monitor to send contextual information to interested parties, an event of a particular type with required attributes needs to be created and then given to the context monitor. The context monitor will then create and send an event by filling in the attributes of the associated event type with contextual information. The decision to use simulated context monitors for generating contextual information was made because of the fact that integrating existing toolkits (e.g., Dey's context toolkit [21]) with PCRA was complex as outside entities must be a Ponder2 managed object. The use of simulated context monitors does not affect our research objectives. In this section we briefly describe some of the context monitors we have implemented that send contextual information regarding various contexts such as user presence, user activity, light intensity, time, etc. to the applications.

5.2.1 User Presence Context Monitor

This context monitor is involved in various example scenarios and simulates user presence by allowing the selection of a user ID and location from its GUI. Once the user ID and

location are selected and a "Send Enter Event" button is pressed, it creates and sends user presence event (user, location) to the application. Figure 5.4 depicts a GUI of this monitor.

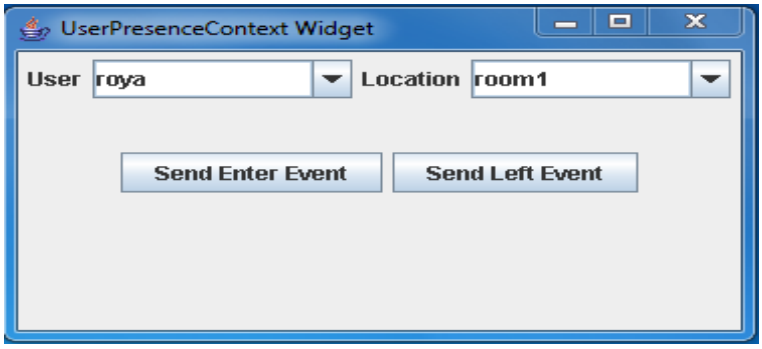

Figure 5.4: User presence context monitor

We have coded a managed object (UserPresenceContext.java) which acts as the user presence context monitor. The following Ponder2 code snippet creates the user presence event type and gives it to the user presence context monitor.

```
(1)
event := root/factory/event create: #("user" "location").
root/event at: "upevent" put: event.

(2)
userpresence := root load: "UserPresenceContext".
root/factory at: "userpresence" put: userpresence.

(3)
root at: "userpresence" put:(root/factory/userpresence event: root/event/upevent).
```

Figure 5.5: Code snippet to create user presence event
and to give it to the presence context monitor

The code highlighted as (1) in figure 5.5 creates the user presence event for the user presence context monitor to send, which has two attributes: user and location. The piece of code highlighted as (2) loads user presence context monitor code and (3) creates an instance of the user presence context monitor and gives it user presence event template created in (1). In the *root/factory/userpresence event: root/event/upevent* in (3), *event* is a factory message of the user presence context monitor, which carries the user presence event template (*root/event/upevent*) as an attribute. This message creates an instance of user presence context monitor and initializes the user presence event template inside the user presence context monitor.

Once the user presence monitor has the user presence event type, and the user and location are selected from its GUI, and the "Send Enter Event" button is pressed, it creates and sends the user presence event through a line of Java code as shown in figure 5.6:

```
anEvent.operation(myP2Object,"create:", P2Object.create(user,location))
```

Figure 5.6: Operation method to create and send the user presence event

anEvent is the event template managed object for the user presence event type, which was given to the user presence context monitor (refer to (3) in figure 5.5) and the *operation* is a method of the P2Object class which performs operations on behalf of basic managed objects. The first argument, *myP2Object* is a Ponder2 managed object for the source of the operation (the user presence context monitor). The second argument *create* is the name of a message, which is one of the annotated methods of the user presence event template. The third argument is a method call *P2Object.create (user, location)* which creates a P2Array of P2Objects for user and location. To summarize, the *operation* method, when invoked, calls the annotated *create* method of the user presence event template which creates and sends a user presence event.

5.2.2 Light Intensity Context Monitor

This context monitor sends data regarding the light value in a given location. It allows inputting an amount of light value in a text field and choosing a location. When the "Send" button is pressed, it creates and sends the light intensity event (light, location) by setting its attributes (light and location) to the amount of light inputted and location chosen respectively. Figure 5.7 depicts a GUI of this monitor.

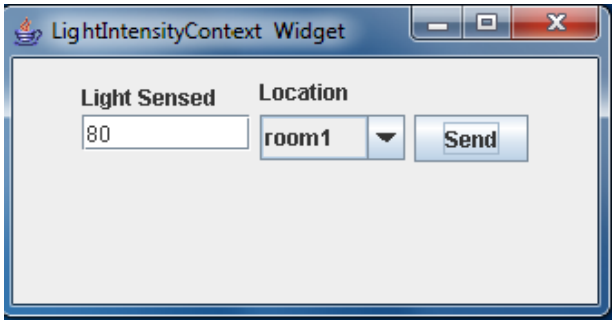

Figure 5.7: Light intensity context monitor

This monitor is coded as a Java managed object (LightIntensityContext.java). This sends the light intensity event through the following line of java code (figure 5.8):

```
anEvent.operation(myP2Object,"create:", P2Object.create(location,light))
```

Figure 5.8: Operation method to create and send the light intensity event

We have discussed the working of the above method call in section 5.2.1 (figure 5.6) and the same discussion applies here except that *anEvent* is the light intensity event template which has two attributes (light and location), and the source of the operation is the light intensity monitor. To summarize, the *operation* method, when invoked, calls the annotated *create* method of the light intensity event template, which creates and sends the light intensity event.

5.2.3 Text Clock

The text clock has functionality that allows a user to set an alarm time and to generate an event when the time reaches the set alarm time or some time (e.g., 10 minutes) earlier than the set alarm time. This was required for implementing Scooby scenario 3 (see section 4.1.2.3in chapter 4) within PCRA. We have coded a Java managed object (TextClock.java), which is a GUI-based text clock and provides said functionality. Figure 5.9 depicts the text clock GUI.

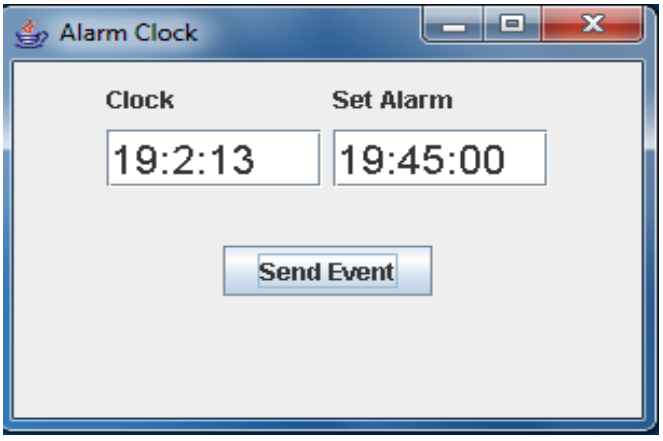

Figure 5.9: Text clock

When a user sets the time in the “Set Alarm” text field and a “Send Event” button is pressed, the text clock creates and sends a time event when the time reaches the set alarm time.

```
event:=root/factory/event create.
root/event at:  "timeevent"  put:event.
root/factory at: "TextClock" put:(root load: "TextClock" ).
root at: "TextClock" put:(root/factory/TextClock event:root event/timeevent).
```

Figure 5.10: Code snippet to create time event and to give it to the text clock

The code shown in figure 5.10 is similar to the one in figure 5.5 that we have discussed except that the monitor involved is the text clock, and the event type associated is the time event type. As can be noted, the event types discussed before (the user presence event type and light intensity event type) have attributes, while the time event type doesn't have any affiliated attributes. We would like to mention here that above code snippet is produced behind the scenes by the utility we have implemented (we discuss this later in section 5.4) to reduce the user written code required for creating an event type, loading and instantiating a context monitor and giving it the event type.

5.2.4 Phone Monitor

The implementation of Scooby scenario 5 within PCRA (see section 4.1.2.5 of chapter 4) needed information about if the phone is ringing, whether a user is attending a call, or whether the user has finished talking (the receiver is placed back on the phone). To send this information we have implemented a Java managed object (PhoneMonitor.java) which is a GUI-based component, and acts as a simulated phone monitor. Figure 5.11 depicts a GUI for the phone monitor.

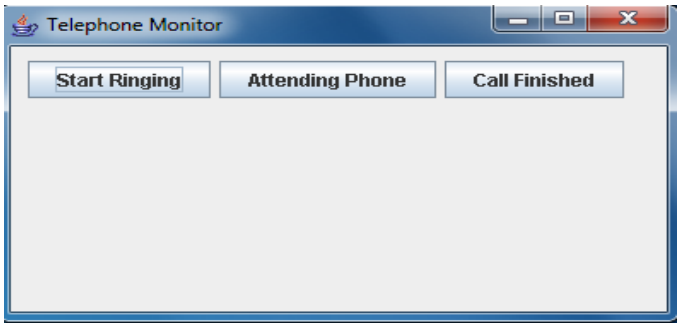

Figure 5.11: Phone monitor

When the “Attending Call” button is pressed, the phone monitor creates and sends a phone event (attendingcall = “true”), signalling that phone is being attended. When the “Call Finished” button is pressed, the phone monitor sends and creates the phone event (callfinished = “true”), signalling that user has finished talking and put back the receiver on the phone.

5.3 Setting up the environment

In order to run the application scenarios on PCRA, the environment needs to be set up. When “*system-utility.p2”* is run from a command prompt with this command *“java net.ponder2.SelfManagedCell –boot system_utility.p2”*, Ponder2 environment is loaded first and then *“system_utility.p2”* is run. This utility includes the code for loading our reconfiguration and adaptation infrastructure in the Ponder2 environment, and the GUI-based system utility. The loading of Ponder2 involves creating domains named policy, event and factory, and importing factory objects of pre-defined managed objects (i.e., EventTemplate, ObligationPolicy, etc.). The policy domain is used to hold policy instances, the event domain is used to hold event instances and the factory domain is used to hold factory objects of managed objects. The next step is to create and add user instances to the system, as discussed before. Once user instances are added, the environment setup is complete and any example scenarios can be executed on PCRA.

5.4 Implementation of the home lighting scenario

In chapter 3, we showed and discussed reconfiguration and adaptation infrastructure system components involved and their interaction in achieving our contributions. We also briefly discussed the implementation of these components. In this section we provide insight into the implementation of our reconfiguration and adaptation infrastructure with the aim of examining and demonstrating our contributions through the implementation of the home lighting example scenario (presented in section 4.1.1.1). For discussing this scenario, we take a corresponding piece of code for each of its parts and discuss what functionality it provides (see section 4.1.1.1 of chapter 4 for the complete source code).

This scenario behaviour involves responding to three contexts: a user presence context, light intensity context and activity context. The part of the code which creates event templates for these contexts and gives them to their corresponding monitors is shown in figure 5.12.

```
(1)
lMAssEvent  lMEvent: #("UserPresenceContext" "userpresence" "user" "location" "enter_or_left").

(2)
lMAssEvent  lMEvent: #("LightIntensityContext" "lightintensity" "location" "lightSensed").

(3)
lMAssEvent lMEvent:  #("ActivityContext" "activityevent" "location" "activity").

(4)
event:= root/factory/event create: #("location").
root/event at: "firstuserevent" put:event.
```

Figure 5.12: Code for the event part of the home lighting scenario

In order to reduce the code required for creating an event type, loading an instantiating context monitor, and giving it created event type, we have implemented a class called "*LoadMonitor_AssociateEvent.java"* which generates the code behind the scenes and loads it dynamically. This component is loaded and instantiated when the system is loaded (see (1) in figure 5.2). The code line highlighted as (1) in figure 5.12 uses this component to create a user presence event (userpresence) with two attributes (user and location) and to give it to the user presence context monitor (UserPresenceContext). The code line marked (2) creates a light intensity event (lightintensity) and gives it to the light intensity context monitor (LightIntensityContext) and code line (3) creates an activity event type (activityevent) and gives it to the activity context monitor (ActivityContext). Figures 5.5 and 5.10, discussed above contain the code for creating the event type, loading and instantiating context monitor and giving it the created event type. As can be noted in figures 5.5 and 5.10, the number of source code lines is 4-5, while using our component for the same task, just 1 line of code is needed. Providing contextual information regarding user presence, light intensity and activity to the home lighting example requires 12 lines of source code without using our component, while using our component, it takes just 3 lines of source code. It can be argued that the use of this component can significantly reduce the code needed when the application has to respond to many contexts. The piece of code highlighted as (4) in figure 5.12 is for creating a first user event type and it has one attribute: location. As we shall see later that the binding policy creates and sends this event and the first user policy responds to this event.

There are seven policies involved in the home lighting example scenario: a *bindingpolicy*, *firstuserpolicy*, *lightpolicy*, *readingpolicy*, *sleepingpolicy*, *watchingtvpolicy* and *userleavingpolicy*. We shall briefly explain each policy in turn and describe the support provided by our reconfiguration and adaptation infrastructure to realize the behaviour of each policy. The code for the *bindingpolicy* is given in figure 5.13.

```
// A binding policy

1.  policy := root/factory/ecapolicy create.
2.  policy event: root/event/userpresence.
3.  policy condition: [:user :location :enter_or_left|
4.     (enter_or_left == "enter")].
5.  policy action:[:user :location |
6.     theUser := users at: user.
7.     configurator createBinding: #(theUser location "LightService").
8.     root/event/firstuserevent create: #(location).].
9.  root/policy at: "bindingpolicy" put: policy.
10. policy active: true.
```

Figure 5.13: Binding policy in the home lighting scenario

This policy subscribes to the user presence context event (line 2). When a user presence event is sent by the user presence context monitor, this policy is executed and one of the messages it invokes is a reconfiguration message (line 7 in figure 5.13) of the reconfiguration manager. This message is mapped into *public void createBinding(P2Object args)* method of the reconfiguration manager as shown in figure 5.14.

```
1. public void createBinding(P2Object args){
2.   try {
3.      P2Object[] p2ObjectArray=null;
4.      p2ObjectArray = args.asArray();
        ….
        ….
        ….
5.    }catch(Ponder2ArgumentException e){
6.       e.printStackTrace();
7.    }
8.  }
```

Figure 5.14: createBinding method of reconfiguration manager

The attributes—*theUser*, *location* and *LightService* in the reconfiguration message (line 7 in figure 5.13) become available in above method (figure 5.14) via the *args* of the P2Object. The *args* of P2Object is converted into an array of P2Object (line 4 in figure 5.14). The *p2ObjectArray* holds all the attributes of the reconfiguration message as P2Objects (i.e., theUser, location and LightService). The method *public void createBinding(P2Object args)* retrieves these arguments from *p2ObjectArray* and interacts with the virtual stub cache manager, one of the system components of the reconfiguration and adaptation infrastructure. The virtual stub cache manager in turns interacts with the registry component to discover a light service (the third element of *p2ObjectArray*) based on contextual information (location of the user—the second element of *p2ObjectArray*), creates a virtual stub instance, initializes it with a real proxy of the discovered light service, caches it locally and then returns it to

createBinding(P2Object args) method. The virtual stub is then handed to the user instance (the first element of *p2ObjectArray*), which means that the binding has been created between the user instance and the light service in particular location represented by its corresponding virtual stub. The detailed description of how adaptive reconfiguration works is given in section 3.2 and also figure 3.9 shows the sequence of messages involved in reconfiguration. In this reconfiguration process, when a service was discovered a virtual stub instance was created, initialized with a real proxy of the found service and cached by the virtual stub cache manager so that the next time this service is required its corresponding virtual stub can be obtained from the cache for improved performance (see sections 3.2 and 3.3 for a detailed discussion). For example, a user A is in room1 and she has a binding with the light service in room1; hence the system has a virtual stub instance for the light service in room1 cached. Now user B enters room1, the user presence context monitor would send and create the user presence event (user = "user B" location = "room1"). In response to this event, the binding policy is executed and the reconfiguration manager (line 7 in figure 5.13) obtains the virtual stub for the light service in room1 from the local cache directly without the need for a remote call, and delivers it to the user instance for user B. This significantly improves the performance of the system.

Another functionality of *public void createBinding(P2Object args)* method, a part of a simple solution to user conflicts, involves maintaining a list of users in a given location (e.g., room1) and sorting this list according to the priority of the users (the older the user, the higher the priority). Whenever any user enters a given location, after bindings are created and delivered to the user instance for her, she is added to the list of users in the given location and then this list is sorted according to the priority of the users. The first user in the list has the highest priority; the second in the list has the next highest priority and so on.

To summarize, the binding policy triggers our reconfiguration and adaptation infrastructure which uses its seamless caching support for improved performance in the process of contextual reconfiguration, and also provides most of the support for simple solution to user conflicts.

The second policy involved in the home lighting example scenario is the *firstuserpolicy,* and its code is shown in figure 5.15.

```
//The first user policy

1.  policy:=root/factory/ecapolicy create.
2.  policy event: root/event/firstuserevent.
3.  policy condition: [:location|
4.     IsFirstUser:=configurator IsFirstUser: #(location).
5.     (IsFirstUser)].
6.  policy action:[:location  |
7.     adaptation performAdaptation:#("LightService" location "on" ).
8.     previousLightValue:= adaptation getRemoteFieldValue:#("LightService" location "getLightLevel").
9.     adaptation performAdaptation:#("LightService" location "adjust" previousLightValue).].
10. root/policy at: "firstuserpolicy" put: policy.
11. policy active: true.
```

Figure 5.15: First user policy in the home lighting scenario

The *firstuserpolicy* subscribes to the first user event and is triggered when this event is sent by the binding policy. The only attribute associated with the first user event is a location. This policy checks with the reconfiguration manager (line 4 in figure 5.15) to determine if the current user in the given location is the first user. If she is the first user in the location, it interacts with the adaptation controller, a system component of the reconfiguration and adaptation infrastructure (discussed in chapter 3) through adaptation messages (see line 7, 8 and 9 in figure 5.15) to turn the light on in the location (e.g., room1), to obtain the last used light value in room1 and then to adjust the light value in room1 to this last used value. The following two messages (figure 5.16) of the adaptation controller, as discussed in chapter 3, form the adaptation interface of PCRA.

```
a. retValue:= adaptation getRemoteFieldValue: #(args)
b. adaptation performAdaptation: #(args)
```

Figure 5.16: Adaptation messages of PCRA

The first adaptation message (a in figure 5.16) has two variants as follows:

```
1. retValue:= adaptation getRemoteFieldValue:#("serName" "loc" "methName")
2. retValue:=adaptation getRemoteFieldValue:#("serName" "loc" "methName" parameter)
```

Figure 5.17: Two variants of getRemoteFieldValue message

The first variant takes three attributes—a service name, location and method name. It invokes a remote method on the remote service and returns a value. For example, in line 8 in figure 5.15, it calls a method ("getLightValue") of a remote service ("LightService") in a

location (e.g., room1) and returns a light value. The second variant has one additional attribute, which is a parameter of the method. It invokes the method on the remote service with a parameter and returns a value. For example, the following line in the home coffee machine, fridge and cooker scenario implementation (see figure 4.12 in chapter 4), invokes the method ("checkAvailability") of the remote service ("FridgeService") in the location ("Kitchen") with a parameter ("milk"), and returns the availability status of the milk in the fridge.

```
isMilkAvail:= adaptation getRemoteFieldValue: #("FridgeService" "kitchen" "checkAvailability" "milk").
```

Figure 5.18: Demonstration of 2nd variant of getRemoteFieldValue message

The second adaptation message (b in figure 5.16) has three variants as follows:

```
1. adaptation performAdaptation:#("serName" "loc" "methName" )
2. adaptation performAdaptation:#("serName" "loc" "methName" parameter)
3. adaptation performAdaptation:#("serName" "loc" "methName" "userPreferredValue")
```

Figure 5.19: Three variants of perfromAdaptation message

The first variant takes three attributes: a service name, location and a method to invoke on the remote service in response to context, performing context-triggered actions. For example, in the first user policy, line 7 (figure 5.15 above) turns the light on when the user entering a particular location (e.g., room1) is the first user. The second variant is used to invoke a method with a parameter on the remote service to modify the behaviour of the service through parameter adjustment. The third variant is a special case where the last attribute is tag information ("userPreferredValue"). This variant internally gets the highest priority user in a given location ("loc"—second attribute), obtains her preferred value for a given service ("serName"—first attribute) and modifies the behaviour of the service by invoking a remote method ("methName"—third attribute) of the service with a parameter (obtained preferred value). The adaptation messages (a and b) are mapped into corresponding methods of adaptation controller, as discussed in chapter 3.

```
public void performAdaptation(P2Object args)
P2Object getRemoteFieldValue(P2Object args)
```

Figure 5.20: PCRA adaptation interface

These two methods of adaptation controller form a generic adaptation interface for PCRA and interact with other components of the reconfiguration and adaptation infrastructure to realize adaptation.

The other policy involved in the home lighting example is the *lightpolicy* and its source code specification is given in figure 5.21.

```
// A policy to modify the light value to the user preferred value if
// the light sensed is lesser than 90% of the user preferred value or
//greater than 110% of the user preferred value.

1.  policy := root/factory/ecapolicy create.
2.  policy event: root/event/lightintensity.
3.  policy condition: [:location :lightSensed |
4.       theUser:= configurator getHighestPriorityUser: #(location).
5.       preferredLightValue :=  theUser getPreferredValue: "LightService".
6.       IsUserSetEnabled:= theUser getExposed.
7.       (IsUserSetEnabled)&(lightSensed < ((90/100)*preferredLightValue))|
         (lightSensed > ((110/100)*preferredLightValue))].
8.  policy action:[:location :lightSensed |
9.       theUser:= configurator getHighestPriorityUser: #(location).
10.      preferredLightValue :=  theUser getPreferredValue: "LightService".
11.      adaptation performAdaptation:#("LightService" location "adjust" "userPreferredValue").].
12. root/policy at: "lightpolicy" put: policy.
13. policy active:true.
```

Figure 5.21: Light policy for the home lighting scenario

The *lightpolicy* policy subscribes to the light intensity event. The light intensity context monitor senses the light in the given location, and then sends and creates a light intensity event (location lightSensed). As a response to this event, the policy gets the highest priority user in the given location from the system. It then checks if the highest priority user is the one for which the light value is already adjusted to her preferred value. If so, it does nothing (leaves the light value unmodified). If the highest priority user is the other user and her light preferred value is less than 90% or greater than 110% of the actual light value, the light value is adjusted to her preferred value.

The policies — *readingpolicy*, *sleepingpolicy* and *watchingtvpolicy* in the home lighting scenario subscribe to the activity event. The activity context monitor monitors the activity of the

highest priority user in the given location and creates and sends an activity event based on what she is doing (e.g., reading, sleeping, or watching TV). For example, when the highest priority user in room1 is reading, the activity monitor creates and sends the activity event (location = "room1" activity = "reading"). In response to this event, all three policies are triggered but the *readingpolicy* is executed. The policy specification code for the *readingpolicy* is shown below, and for the policy specification code of the *sleepingpolic y* and *watchingtvpolicy*, see figure 4.2 of chapter 4.

```
// The policy to modify the behaviour of light service when the user
// is reading.

1.  policy := root/factory/ecapolicy create.
2.  policy event: root/event/activityevent.
3.  policy condition: [:location :activity |
4.       (activity=="reading")].
5.  policy action: [:location :activity |
6.       theUser:= configurator getHighestPriorityUser: #(location).
7.       prefValReadActivity :=  theUser getPreferredValue: "LightService_reading".
8.       adaptation performAdaptation:#("LightService" location "adjust" prefValReadActivity).].
9.  root/policy at: "readingpolicy" put: policy.
10.  policy active: true.
```

Figure 5.22: Reading policy in the home lighting scenario

When executed, the *readingpolicy* policy gets a user preferred value for the light service for the reading activity and modifies the light value in room1 to this value. When the user in room1 is sleeping, the activity event (location = "room1" activity = "sleeping") is created and sent. In response to this event, the *sleepingpolicy* is executed and obtains the user preferred light value for the sleeping activity, and then modifies the light value to this value. Similarly, when the user in room1 is watching TV, the activity event (location = "room1" activity = "watchingtv") is created and sent. The *watchingtvpolicy* is executed and gets the user preferred light value for this activity and then modifies the light value to this value.

The last policy in the home lighting scenario is the *userleavingpolicy* and the policy code specification is shown below.

```
// The user leaving policy

1.  policy := root/factory/ecapolicy create.
2.  policy event: root/event/userpresence.
3.  policy condition: [ :user :location:enter_or_left|
        (enter_or_left=="left")].
4.  policy action: [ :user :location :enter_or_left |
5.       configurator removeBinding: #(user location).
6.       adaptation performAdaptation:#("LightService" location "adjust" "userPreferredValue").].
7.  root/policy at: "userleavingpolicy" put: policy.
8.  policy active: true.
```

Figure 5.23: The user leaving policy in the home lighting scenario

The *userleavingpoliy* subscribes to the user presence event (line 2). Let us suppose there are three users in room1, Maanta, Yasir and Roya, and the light value in room1 is adjusted to Roya's preferred value as she is the highest-priority user. Now Roya leaves room1. The user presence context monitor detects her leaving, and creates and sends the user presence event (user = "Roya" location = "room1" enter_left = "left"). In response to this event, the reconfiguration manager (line 5) removes her from the list of users in room1 and then sets a list of users so that the next highest priority user moves to the top of the list and becomes the highest priority user. The user, Yasir was second in the list of users (the next highest priority user) in room1 before Roya was removed and now Yasir has become the highest priority user. The code (line 6) internally gets the highest priority user, obtains his preferred value for the light and adjusts the light level in room1 to his preferred value. If the user who left room1 is not the highest user, this user will simply be removed from the list and the light value in room1 will remain unmodified.

5.5 Summary

In this chapter, we described the features and discussed the implementation details of our GUI-based system utility. This utility provides functionality that allows performing various tasks through its GUI. These include creating and adding user instances to and removing from the PCRA environment without stopping or interrupting the system; allowing end users to customize their preferences for various services dynamically, and allowing the loading and executing of example scenarios through its GUI.

We also discussed briefly the implementation of various context monitors which provide simulated contextual information regarding various contexts involved in the example scenarios. We also discussed implementation details of our reconfiguration and adaptation infrastructure through the implementation of the home lighting example scenario, and demonstrated our contributions supported by our infrastructure.

We also discussed very briefly how to set up the PCRA environment to be able to run example scenarios on it. *"system_utility.p2"*, in addition to other code, includes code for loading our reconfiguration and adaptation infrastructure into Ponder2 environment. When the "*system_utility.p2"* file is run through the command *"java net.ponder2.SelfManagedCell –boot system_utility.p2"*, it first loads the Ponder2 environment, and then runs *"system_utility.p2"*, loading our reconfiguration and adaptation infrastructure and the system utility in the Ponder2 environment. The source code of PCRA system, including example scenarios has been made available at http://pcra.usindh.edu.pk/.

Chapter 6

Evaluation

The main argument of this thesis is that a policy-based programming model provides an effective means for developing, modifying and extending adaptive context-aware applications. The goal of this chapter is to evaluate the effectiveness of our policy-based approach at developing adaptive context-aware applications. In this chapter we also study the performance of two main features of our proposed system, PCRA: policy-based contextual reconfiguration and policy-based contextual adaptation. The advantages offered by our approach are simplification of development effort, dynamic modification and dynamic extensibility of adaptive context-aware applications, and support for user involvement. In order to evaluate the effectiveness of our approach, we divide the evaluation process into three categories: High-level Analysis, Qualitative Evaluation and Performance Evaluation. The high-level analysis evaluates the first aspect of effectiveness of our approach — simplification of development efforts, while qualitative evaluation evaluates other aspects of the effectiveness of approach — dynamic modification, dynamic extensibility and the support for user involvement. In our performance evaluation, we study the performance of policy-based reconfiguration and adaptation where we conduct various tests to determine reconfiguration and adaptation time under both local and distributed settings.

In the high-level analysis and qualitative evaluation, we evaluate our approach and compare it with the Scooby [23,69], a specifically designed service composition language. Our work and Scooby share similar research goals in that we both advocate the use of high-level means to achieve service composition / reconfiguration to simplify the development task. However, we use a different approach to achieving this. Scooby's main idea is that a dedicated domain specific language is a more effective way of performing service composition in which composed services can be developed using high-level binding directives to discover and bind services rather than traditional approaches that use an API, whilst in contrast we advocate that the use of a policy-based programming model provides more effective means for carrying out context-aware reconfiguration. Scooby has compared their approach to One.World [76-78,84] by implementing five example scenarios on both Scooby and on One.World and using them as the basis for comparison. In order to compare our approach to both Scooby and One.World, we follow the same comparison methodology and thus have implemented the same example

scenarios (see chapter 4) on PCRA. In the high-level analysis, we use the same tables and metrics structure as used by Scooby to compare their work with One.World, and investigate the effectiveness of PCRA, Scooby and One.World with regard to reduced development efforts. The qualitative evaluation section evaluates PCRA and Scooby with regard to other aspects of effectiveness of an approach: dynamic modification, dynamic extensibility and the support for user involvement.

The low-level analysis in Scooby includes various test comparisons to evaluate the effectiveness of the Scooby middleware layer with regard to event scaling, event latency, advertisement latency, late binding latency and a number of services. Scooby uses an event-based mechanism as a primary form of internal communication and thus is developed upon the Elvin notification system[1]. Every form of communication in Scooby such as registration and discovery of the services, remote method invocation, etc. is event-based and hence the low-level analysis in Scooby includes event scaling and event latency comparison tests. Although our research goals are similar to the Scooby in that we advocate and use high-level means through the use of policy specifications for reconfiguration, while Scooby provides high-level constructs in the form of binding variables, both systems are conceptually different at the design level. As our design involves polices and polices use events, this is where the event-based communication takes place and the rest of the design does not use events as means for communication between various components of the system. Moreover, we use Ponder2 for specifying polices and Ponder2 has its own event model. Therefore, we don't consider comparison tests such as event scaling and latency relevant in the context of our system.

6.1 High-level Analysis

This part of our analysis in combination with the qualitative evaluation part in section 6.2 determines if taking a policy-based approach for developing, modifying and extending adaptive context-aware applications is more effective than the use of a specifically designed service composition language (Scooby) and that of an API based approach (One.World). We follow the analysis and points of comparison made by Scooby against One.World.

6.1.1 Language comparisons

One of the comparisons made by Scooby was language principles. We use the same table as given in Scooby thesis [69] and add our system (PCRA), and examine how the three systems compare with respect to common traits and concepts they share.

[1] Elvin router is no longer available, but Avis router [99] is a completely compatible substitute of Elvin router.

Metric	Scooby	One.World	PCRA
Binding variables	√	√[2]	√
Service Discovery	√	√	√
Service Alteration	√	x	x
Service Descriptions	√	x	x
Remote Invocation	√	x	√
Object Oriented	√	√	√
Keyword Count	74	20 packages,353 classes, rest of java language	79[a]
Event Handling	√	√	√[b]
Java Extension(API)	x	√	x
Method Delegation	√	x	x
Inheritance	x	Single	x
Typing	Static	Static	Dynamic
Exception Handling	Bindings	√	√[c] (Virtual Stub)
Arithmetic Handling	√	√	√
Event Driven	√	√[3]	√
Global Variables	√	√[4]	x
Procedural/Declarative	Procedural	Procedural	Declarative

Table 6.1: Language comparisons

Any data in table 6.1 for Scooby system and One.World, and any footnotes regarding One.World are reproduced from Scooby [69]. The superscript marks a, b and c, in the column for PCRA are defined below.

(a) PCRA is built on top of Ponder2. In Ponder2, as discussed in chapter 2, everything is a managed object (e.g., Event, Policy, Domain, etc.), and is accessed through message keywords. As a result, the message keywords of all managed objects are the number of keywords in the Ponder2 system. The documentation with all the Ponder2 keywords can be found at: http://www.ponder2.net/doc/pondertalk/ which lists 224 keywords, including factory messages of all managed objects. The PCRA uses a small set of Ponder2 keywords (e.g., create, event, condition, action, active, at: , at:put: , load: print:, remove:, +, - < ==, !=, &, NOT, etc.). These keywords are the ones which are needed for basic operations such as specifying policies, events, performing some basic domain operations, boolean operations, string operations. PCRA uses total of 79 keywords. 27 of them are Ponder2 keywords, while the remaining 52 are message keywords of the managed objects which we have implemented to build our reconfiguration and adaptation infrastructure.

[2] Binding variables found in One.World differ considerably to those found in Scooby. However, the terminology used is the same.

[3] One.World provides an API that augments Java with a way to interact with the environment. Used within this context, One.World is able to achieve similar functionality to that found in Scooby, but requires additional programming to accomplish this.

[4] Global variables can be achieved by using the Java Language. However, these do not strictly adhere to the same definition as found within Scooby.

(b) PCRA supports event handling in which a policy is an event handler which is triggered in response to an event to which it is subscribed. This is where the event is used as a means of communication and the rest of the communication between system components or with remote bound services does not involve the use of events. However, within Scooby, as discussed before, every form of communication is event-based.

(c) PCRA supports exception handling in general because both Ponder2 and our reconfiguration and adaptation infrastructure are implemented in Java. However, one of our contributions is the management of bindings where the virtual stub is in charge of handling an exception when a bound service becomes invalid. This reduces the need for a common class of exception handling in application/deployment code.

From the above table we observe the following:

- PCRA, Scooby and One.World all are event-driven object-oriented systems. However, the PCRA is a policy-based declarative object-oriented system, while both Scooby and One.World are procedural object-oriented systems. In PCRA, the program functionality is implemented through policies and policies contain reconfiguration and adaptation messages to perform reconfiguration and adaptation. As the policies are specified declaratively, the complexity involved in developing adaptive context-aware applications is reduced. Similarly, Scooby provides high-level binding constructs. In Scooby, the main functionality of any composed services is bindings to services and events, which is implemented through the use of these high-level abstractions, thus contributing towards reducing the complexity involved in developing adaptive applications. However, the other parts of the Scooby code are written in syntax similar to Java, unlike the PCRA where all application functionality is specified declaratively through policy specifications. Within One.World, applications are coded using an API, while PCRA and Scooby provide user-level abstractions to API.

- PCRA is a dynamically typed system, while both Scooby and One.World are statically typed systems. Both approaches have trade-offs. For example, the statically typed systems can find type errors at compile time, thus increasing the reliability of the programs (program correctness). Moreover, the compiled-code executes more quickly as its execution does not involve type checking. However, from the development perspective, static typing requires developers to learn and follow the type system, introducing additional burden on the developer. In contrast, the dynamically typed systems allow typing checking at runtime, hence may result in runtime errors. However, from the development perspective, it allows programmers to think and develop programs at a higher level in the sense that they are not required to concern about the types of data. As the main argument of

PCRA and Scooby is a provision of high-level means to simplify the development task, we argue, in light of above discussion, that PCRA being dynamically typed contributes better towards this goal than that of Scooby and One.World.

- The number of keywords within the PCRA is a slightly higher than that of Scooby. This is due to the fact that the small set of Ponder2 keywords (27) used in PCRA includes keywords which are related to policy (e.g., condition, action, event, active), domain operations (e.g., at, at:put:, remove) and factory messages of policy, event and domain managed objects. As compared to the PCRA and the Scooby, One.World includes a large number of classes (353) and each of which has subsequent methods, hence a more flexible programming model. This flexibility is achieved at the cost of ease of use and complexity, as argued in the Scooby thesis. The reduced syntax of both PCRA and Scooby brings the ease of use and less complexity.
- Both PCRA and Scooby support remote invocations, but this is not present within One.World. In Scooby, remote invocations are accommodated through the use of binding variables, whilst, within PCRA, the remote invocations are achieved through adaptation messages.
- Scooby provides features such as service alteration, service characteristics, etc. However, the PCRA does not offer such features because the PCRA is intended to provide the policy-based support for reconfiguration and adaptation, not writing or describing services.

6.1.2 Source code specification lines

The purpose of the following comparison is to examine PCRA, Scooby and One.World systems with respect to how much effort is needed to code adaptive context-aware applications. This effort is measured in terms of source code lines needed to do so. All five Scooby scenarios (the simple printer service composition, follow me service, home coffee machine, fridge and cooker, home environment and music & telephone scenario) discussed in chapter 4 are used to note the number of lines needed to code each of these under all three systems.

Metric	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Scenario 5
PCRA(lines)	27	17	41	14	32
Scooby: Composed Service(lines)	15	23	41	15	27
Scooby: All Services(lines)	44	62	72	88	91
One.World(lines)	366	712	728	724	549

Table 6.2: Source code specification comparisons

We assume the fact that services would be available in the environment and the developer/end user would only be required to code policy specification or to develop a composed service to discover the required services and to bind them. Considering this assumption and also the main argument of Scooby that service composition can be achieved efficiently by using a dedicated composition language, we compare the Scooby code specification for the composed service rather than all the services (the composed service and the ones that the composed service would discover and bind to) with its counterpart policy specifications in the PCRA. The Scooby source code for composed services for all the example scenarios and the counterpart PCRA specification code can be found at in chapter 4, for comparison. The Scooby code for composed services for all scenarios can also be found in chapter 6 of Scooby thesis [69].

Recall that the numbers for One.World source code lines (table 6.2) for all example scenarios have been taken and reproduced from Scooby thesis [69]. The number of source code lines in each scenario is the total of both the source code lines of the composed service and the source code lines of the services the composed service is composed of. We don't have the One.World source code of example scenarios to separate out the source code lines of the composed service. However, as can be noted, the number of source code lines for each of the example scenarios (all services) in One.World is far higher than the Scooby source code (all services). For example, the number of the One.World source code lines in example scenario 3 is 712, while the number of the Scooby code source lines is 62. This gives enough evidence to assume that the number of One.World source code lines for the composed service in each of example scenarios would be higher than the number required by Scooby for corresponding composed service.

The numbers in the table 6.2 clearly indicate the amount of effort needed to code the scenarios using an API based approach (One.World) is a far greater than that of a policy-based approach (PCRA) or a specifically designed service composition language (Scooby). However, when comparing PCRA and Scooby, the table figures show the amount of efforts needed to code the scenarios is arguably similar.

6.1.3 Expressiveness

In this section, we examine the expressiveness of PCRA against Scooby and One.World with regard to the concepts (binding variables, remote invocation, event handling, etc.) discussed in section 6.1. We use the same set of criteria formulated and used by Scooby to compare with One.World. Therefore, we add a column for the PCRA in the expressiveness table used by Scooby and the resulting table 6.3 is shown below. The data for Scooby and One.World

systems in table 6.3 is the same as in original expressiveness table found in Scooby [23] except an additional metric, which is the "Lines: composed service". In this, we have separated out the source code lines of the composed service in each example scenario developed using Scooby from the total number of lines, which is the total of both the number of source code lines in the composed service and other services that the composed service is composed of.

In table 6.3, the column 'A' represents Scooby coding and 'B' One.World equivalent, and 'C' PCRA equivalent.

	Scenario 1			Scenario 2			Scenario 3			Scenario 4			Scenario 5		
Metric	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
Number of services	3	2	2	3	3	3	4	4	3	4	4	4	3	3	2
Lines: Composed service[5]	15	-	27	23	-	15	41	-	41	15	-	13	27	-	32
Lines: All services	44	366	27	62	712	15	72	728	41	88	724	13	91	549	32
Variable declarations	3	22	0	0	41	0	2	56	0	1	44	2	5	72	4
Binding constructs[6]	2	2	5	5	7	3	5	9	5	5	8	4	4	7	4
Binding attributes	2	2	4	4	0	3	3	0	4	3	0	4	4	0	4
Binding variables	1	1	2	4	0	3	0	0	1	0	0	1	0	0	1
Method declarations	3	20	2	3	40	4	4	40	3	9	40	10	10	30	7
Remote invocations	2	0	3	2	0	1	10	0	7	10	0	4	11	0	8
Dynamic characteristics	3	0	0	2	0	0	0	0	0	0	0	0	0	0	0
Static characteristics	3	0	0	3	0	0	4	0	0	2	0	0	3	0	0
Event handlers	0	4	3	1	8	2	0	8	4	1	8	1	4	6	3

Table 6.3: Expressiveness comparisons

In Scooby, in the context of above table, the binding variables are the dynamic attributes that allow binding constructs to evolve over time. However, the binding variable in the Scooby thesis is explained as a construct which binds to a service. Scooby provides two forms of binding constructs: a binding that binds to a service for performing remote method invocation on this service and this is called the service binding, and another that allows the service to capture an event.

Within PCRA, the service bindings are resolved when the binding policy is triggered in response to an event, unlike in Scooby in which service bindings are resolved when the composed service is executed. As a result, as mentioned in footnote 6, an event binding is associated with each policy; there would be one additional binding construct involved in each of the scenarios developed using PCRA.

[5]As the analysis of One.World is done by the implementer of the Scooby, we don't have source code of the example scenarios to separate out the source code lines of the composed service involved in each of scenarios, hence entries are left blank.

[6]In Scooby, the binding constructs count involves both service bindings and event bindings. In PCRA, each policy responds to an event and thus has a specification to capture an event (e.g., policy event: root/event/eventtype). This specification can be considered to correspond to an event binding in the Scooby. Therefore, the number of policies in the example scenarios tells the number of event bindings in each scenario. The reconfiguration message in PCRA corresponds to a service binding in the Scooby. Therefore, the binding constructs count in PCRA is a total of the number of policies and reconfiguration messages.

The number of binding constructs involved in all scenarios in both the PCRA and the Scooby is more or less the same except in example scenario 1 in which the number of binding constructs for PCRA is 5 and for Scooby only 2. This is due to the fact that (1) there is a binding policy (resulting in one event binding) in which bindings to the printer and converter service are resolved and (2) the document and its type are dynamically becoming available through an event. There are two policies (hence two event bindings) that respond to that event—one policy checks the document type and if the type is 'pdf' it sends the document to the printer, and other policy checks if the document type is not 'pdf' it uses the converter service to convert the document into pdf format and sends the converted document to the printer. However, there are no event bindings involved in the Scooby code for this scenario because (1) bindings with both printer and converter service are resolved when the composed service is executed (hence no event binding), and the document and its type are assumed to be available in the code and hence have no event bindings.

Binding Constructs	**Scenario 1**		**Scenario 2**		**Scenario 3**		**Scenario 4**		**Scenario 5**	
	PCRA	**Scooby**	**PCRA**	**Scooby**	**PCRA**	**Scooby**	**PCRA**	**Scooby**	**PCRA**	**Scooby**
Service bindings	2	2	1	3	1	5	3	4	1	2
Event bindings	3	0	2	2	4	0	1	1	3	2

Table 6.4: Binding constructs comparisons

The total number of binding constructs is the sum of both the number of service bindings and event bindings. There are more event bindings involved in all scenarios developed using PCRA due to the use of policies because there is an event binding associated with each policy. Even though there are more event bindings involved, the total number of binding constructs within Scooby and PCRA is almost the same. This is due to the fact that fewer service binding constructs are involved within PCRA (refer to table 6.4). The reason for this is that, within PCRA, a single reconfiguration message (service binding construct in Scooby terminology) can discover and bind to more than one service if the search criterion for these services is the same. For example, discovering both the music service and the light service in a living room and binding them can be expressed within the PCRA with a single reconfiguration message as shown in figure 6.1.

```
createBinding: #(theUser location "LightService" "MusicService")
```

Figure 6.1: Expressing a reconfiguration message in PCRA

Note that the search criterion for both the services is the same, i.e., location (living room). However, within Scooby, there has to be a separate binding variable for resolving each service and for the above situation two binding variables have to be coded as follows (figure 6.2):

```
service telephone decorates music: cdplayer, lighting: lightingcontroller {
    bind cd match { location: "livingroom" }
    bind lighting match { location: "livingroom" }
```

Figure 6.2: Expressing service bindings in Scooby

This may prove costly if there are many other services in the same living room to be discovered and bound to, such as a TV service, curtain service, smart screen, etc., and the search criteria is the location (living room) only. Within PCRA, what is needed to be done is to add the service names of these services in the reconfiguration message (no additional reconfiguration message) as under (figure 6.3):

```
createBinding: #(theUser location  "MusicService" "LightService"
            "TVService" "CurtainService" "Screen")
```

Figure 6.3: A single PCRA reconfiguration message expressing multiple services with the same search criteria

Whereas within the Scooby, there have to be three binding variables coded each for three services, resulting in three more service bindings. The total number of service bindings would be 5 as shown in figure 6.4.

```
service telephone decorates music: cdplayer, lighting: lightingcontroller,
tv: tvservice, cs: curtainservice, sc: screen {
    bind music match { location: "livingroom" }
    bind lighting match { location: "livingroom" }
    bind tv match { location: "livingroom" }
    bind cs match { location: "livingroom" }
    bind sc match { location: "livingroom" }
```

Figure 6.4: The Scooby code expressing multiple services with the same search criteria

This clearly shows that the core concept of service bindings can be expressed more efficiently within PCRA than within Scooby.

The remote invocation support within PCRA is a fundamental part of policy-based contextual adaptation support in which adaptation messages are used to perform remote method invocations on bound services to modify the behavior of these services in response to policy evaluation. There are two types of adaptation messages within PCRA (discussed in chapter 5) — one that performs remote invocations on the bound services in order to access their remote fields and another that performs remote method invocations on the bound services to achieve contextual triggered actions and modification of these actions through parameter adjustments. Within Scooby, the remote invocations are accommodated through binding variables in which a method invocation on a remote service takes place through its corresponding binding variable, and the syntax for this is the same as in Java (e.g., instance.methodName (args)).

Scooby offers both static and dynamic service attributes. However, PCRA does not offer such features as it is primarily aimed at developing adaptive context-aware applications through policy specifications, not writing services and their descriptions. Therefore, entries for static and dynamic service characteristics in all scenarios have 0 values. However, PCRA considers and addresses other important surrounding issues such as caching virtual stubs for improved performance and providing a very simple solution to personal conflicts.

To summarize, the figures in table 6.3 for all example scenarios suggest that PCRA and Scooby are more expressive than One.World with regard to common concepts (e.g., binding constructs, event handlers, etc.) that all three system share. This fact is supported by the number of source code lines involved in implementation of example scenarios under three systems (see table 6.2). The detailed analysis of One.World for the comparison with Scooby can be found in the analysis chapter of Scooby thesis [69]. The expressiveness comparison between PCRA and Scooby is hard to make as the number of various constructs involved in each of scenarios is more or less the same (hence the source code lines in each of scenarios). However, with respect to some important features, such as policies being declarative and the expressiveness of reconfiguration messages (service bindings in Scooby terminology); we argue that the PCRA is more expressive than Scooby.

Based on the results of various tables and arguments presented in this part of analysis, we argue that both PCRA and Scooby provide effective means of developing adaptive context-aware applications in the form of a high-level programming model. To further evaluate the effectiveness of our approach, we consider other important aspects of adaptive context-aware applications: modifiability, extensibility and the support for user involvement. In next section we evaluate these parameters and compare with Scooby.

6.2 Qualitative Evaluation

In the previous section we evaluated an effectiveness of an approach used within PCRA with respect to simplification of development efforts and compared it with Scooby and One.World. In this section we further evaluate the effectiveness of our approach with respect to other important parameters: modifiability, extensibility and support for user involvement, and investigate how it compares with Scooby.

6.2.1 Modifiability

One of the important aspects related to adaptive context-aware applications is their modifiability. The modifiability may be considered as related to modifying the existing adaptive behaviour of the application. The application may evolve over time, requiring modification. For example, user preferences for different service may change over time or the applications may require modifying their existing adaptive behaviour, for example, the current behaviour of the applications is that when a user leaves the room, the light is turned OFF, and now it is desirable that light should not be turned OFF but the light value be set to a dim value instead. We take the example scenario 5, the telephone & music scenario (discussed in chapter 4) and slightly modify it, and then show how this modification can be achieved within both the PCRA and the Scooby. We reproduce the description of this scenario below.

"*The user is sitting in a dimly lit living room, listening to music. Suddenly the phone starts ringing. Connected to the phone is a device that detects an incoming call. This automatically causes the lighting levels to be returned to normal (if previously dimmed) and for the volume in the CD player to be reduced. The user picks up the phone and begins to talk. Once the conversation has finished, and the receiver is placed back on the phone, the lighting is returned to the previous levels, and the volume returns to what it was before the phone call."*

We pick up this scenario from the point that when the phone is ringing and the call is attended, the lighting levels are returned to normal and the volume in the CD player is reduced. Below (figure 6.5) is the Scooby code for the functionality of this part, the complete code listing can be found in chapter 4. The name of the composed service which provides the functionality of the this scenario is *telephone.s*

```
1. on notification( phoneInUse ) {
2.    volumeLevel = cd.getVolume()
3.    lightingLevel = lighting.getBrightnessLevel()
4.    cd.setVolume( 0 )
5.    lighting.turnLightsOn()
6.    }
```

Figure 6.5: The part of the Scooby code for telephone & music scenario

The above code snippet (figure 6.5, line 4) shows when the call is being attended; the volume of the CD is reduced to 0 units. We slightly modify this behavior that the volume in the CD player is reduced to 10 units instead of 0. In order to make this slight modification in the application behavior, the following steps need to be performed:

- The whole application should be stopped.
- The line 4 in *telephone.s* file should be changed to *cd.setVolume (10).*
- The modified *telephone.s* file should be recompiled using a Scooby compiler and this would produce the corresponding *telephone.java* file.
- The *telephone.java* file should be compiled with a Java compiler and this would produce *telephone.class*.
- The *telephone.class* should be re-run.

This can be noted from above steps that in order to carry out this modification, the application was stopped, recompiled and restarted, thereby not allowing dynamic modification of the application. We show how this modification can be achieved within the PCRA. The following code snippet (figure 6.6) is the *attendingcall* policy, the counterpart of Scooby coding in figure 6.3. The complete code listing of the scenario can be found in chapter 4.

```
1.  policy := root/factory/ecapolicy create.
2.  policy event: root/event/phoneEvent.
3.  policy condition: [:attendingCall :callfinished |(attendingcall)].
4.  policy action: [
5.      musicVol:= adaptationgetRemoteFieldValue:#("MusService" "livingroom" "getVolume").
6.      lightVal:= adaptation getRemoteFieldValue:#("LightService" "livingroom" "getLightLevel").
7.      variableSaver setLightValue: lightVal.
8.      variableSaver setMusicVolume: musicVol.
9.      adaptation performAdaptation:#("LightService" "livingroom" "adjust" 70).
10.     adaptation performAdaptation:#("MusicService" "livingroom" "setVolume" 0).].
11. root/policy at: "attendingcall" put: policy.
12. policy active: true.
```

Figure 6.6: The attending call policy in the telephone & music scenario

In order to carry out the same modification within the PCRA, the following steps need to be performed:

- Change the line 10 in the policy specification in figure 6.4 as under: *adaptation performAdaptation: #("MusicService" "livingroom" "setVolume" 10).*
- Load the modified policy into the system. The modified policy can be loaded dynamically through our GUI-based utility (discussed in chapter 5).

It can be noted from above steps that modification is achieved dynamically without stopping, recompiling and restarting the application. In One.World, the applications have to be coded in Java and their modification also require going through the same steps as in Scooby— stopping, recompiling and restarting the application.

To summarize, PCRA allows dynamic modification of applications, while, within Scooby and One.World, the application needs to be stopped, recompiled and restarted. Hence PCRA outperforms Scooby and One.World by being able to make dynamic, rather than static, modifications to policies.

6.2.2 Extensibility

Another important aspect of adaptive context-aware applications is their extensibility where they are required to be extended in terms of responding to additional contextual triggers which were not foreseen when they were initially developed. For example, the light music and telephone scenario discussed above initially responded to a telephone contextual trigger (phone being attended/phone receiver placed back)— when the user attended the call, the system caused the light level to be returned to normal and for the volume in the CD player to be reduced. Once the user finished talking and the receiver was placed back on the phone, the system modified the light value back to the previous level and the volume to what it was before the phone call. Now it is desirable that the application should be extended so that if the user does not wish to attend the call within some time period (e.g., 5 seconds) after the phone started ringing, a voice message on an answering machine is recorded. This extension requires adding a contextual trigger, time and the behavior that would get triggered in response to time context, allowing the voice message to be recorded on the answering machine.

We have shown before that any change of behavior in the applications developed using Scooby or One.World requires going through the steps involving stopping, recompiling and restarting the application. However, we show and discuss how this extensibility is achieved within PCRA. The following steps need to be performed to extend the example scenario.

- The time event type needs to be created, the timer to be loaded and then the time event to be given to the timer. The timer would receive a signal from the telephone context monitor that detects the incoming call (ringing) and whether the call is being attended or the phone receiver has been placed back. As soon as the phone starts ringing, the timer receives the signal from the telephone context monitor and starts noting the time.
    - If the call is not attended within 5 seconds, the timer creates and sends a time event. In response to the time event, the policy would get triggered allowing the voice message to be recorded on the answering machine.
    - If the call is attended within 5 seconds, the telephone monitor signals the timer to stop noting the time.

  The following code snippet creates the time event type, loads the timer and gives it the time event type.

```
1. event:=root/factory/event create.
2. root/event at:  "timeEvent"  put:event.
3. root/factory at: "Timer" put:(root load: "Timer" ).
4. root at:"Timer" put:(root/factory/Timer event: root/event/timeEvent).
```

Figure 6.7: The code snippet for creating a time event template, loading the timer and giving it the time event type.

- This piece of code can be loaded dynamically into the PCRA through our GUI-based system utility. As a result, a new contextual trigger, time will be loaded into the system dynamically.
- The remaining part is to specify and load the behavior that would get triggered in response to time context. The following code snippet (figure 6.8) is a policy which specifies the required behavior and responds to the time event.

```
policy := root/factory/ecapolicy create.
policy event: root/event/timeEvent.

policy action: [
   adaptation performAdaptation:#("AnsMachine" "livingroom" "recordMsg" "msg").].

root/policy at: "timepolicy" put: policy.
policy active: true.
```

Figure 6.8: The time policy to record a message

- This policy can dynamically be loaded into the system and would respond to time context sent by the timer when it has been 5 seconds since the phone started ringing.

It can be noted from the above discussion that the extension to the music and light scenario in terms of adding the additional contextual trigger and the required behavior was carried out dynamically without shutting down or interrupting the system. However, dynamic extensibility is not possible within Scooby and One.World since it would require going through stopping, recompiling and restarting the application.

6.2.3 User Involvement

The importance of user involvement in specifying and reconfiguring the behavior of adaptive context-aware applications has been advocated in various research efforts, e.g., [2,22, 79,100,101]. Our system design supports end users to specify and modify the behaviour of adaptive context-aware applications. The system design includes policies for specifying adaptive behavior of the applications, and the user component which models the user of the environment. The user component in addition to other functionality, allows the users to set their preferences for different services. As discussed (in chapter 5), the system utility we have developed allows the end users to customize their preferences for different services through a GUI.

There may come times when the user preferences may change and the user may want to reconfigure the behaviour of the application accordingly. For example, in a simple home lighting example when the user enters the room, the light is turned ON and the light value is adjusted to her preference (for example, 80 units). Now the user preference for the light service has changed and she would want the light value to be adjusted to 100. In order to customize this behaviour, all the user needs to do is to change her preference for the light service using our GUI-based system utility without any change in the code and this does not require any special technical skill or abilities. However, there may come times when new behaviours may need to be specified and added or the modification in behaviour may require more than setting up a preference value. As the applications are to be developed using polices on the PCRA, adaptive behaviours involved in the applications are to be specified using policies and a policy is independent unit of execution. This means that complete adaptive context-aware application would be comprised of polices and each policy implements a particular adaptive behaviour. We shall see how this feature of policy contributes towards supporting user involvement. In addition, polices are expressed declaratively unlike procedural languages. These features of policies — being declarative, the policy implementing a particular adaptive behavior and an independent unit of execution support user involvement. For example, in section 6.2.1, we saw

that in order to modify the behaviour of the application, the change was only made in that policy that was implementing the behaviour which was required to be modified and then was loaded dynamically. Similarly, in order to add an additional adaptive behavior into an application, the policy needs to be specified implementing this behaviour and loaded dynamically. Doing so does not require the user to be a technical person. However, this may require providing a little training on how to express policies using high-level specifications and a little knowledge of how to dynamically activate, deactivate, load and unload policies.

Although Scooby provides high-level constructs and thus simplifies the task of developing adaptive applications, this approach still requires the user to have a fair amount of technical knowledge as the user has to specify not only what to do but also how to do it. Besides this, as discussed before, to modify the existing adaptive behavior or to add the additional adaptive behavior requires going through the process of stopping the application, (re)compiling the Scooby code using Scooby compiler, (re)compiling the Scooby compiler generated Java code and then restarting the application (i.e., requires fair amount of technical training to the user for this process). In our high-level analysis (section 6.1), it was evaluated that both PCRA and Scooby provide high-level means to simplify the development task, while, within One.World, the applications are developed using an API (i.e., requires knowing low-level details of the underlying system, making application development hard for the user). As a result, One.World does not support user involvement in specifying and reconfiguring the behavior of adaptive context-aware applications. This clearly indicates that Scooby provides better support for the user involvement than its counterpart, One.World. However, in the light of arguments presented above, we argue that PCRA provides better support for the user involvement than Scooby.

6.3 Performance Evaluation

We studied the performance of two main features of PCRA: policy-based contextual reconfiguration and policy-based contextual adaptation. Tests conducted include reconfiguration time (binding time) and adaptation time under both a local setting and distributed setting. The reconfiguration time is the time taken by PCRA when creating a binding between a user instance and a remote service in response to policy evaluation. The adaptation time is the time taken by PCRA to adapt the behavior of the service involved in the binding in response to policy evaluation.

The reconfiguration time is measured from a point context is sent by the context monitor to the policy, which has subscribed to it, until the binding is established in response to policy evaluation. Figure 6.9 shows sequence diagram for reconfiguration time without cache.

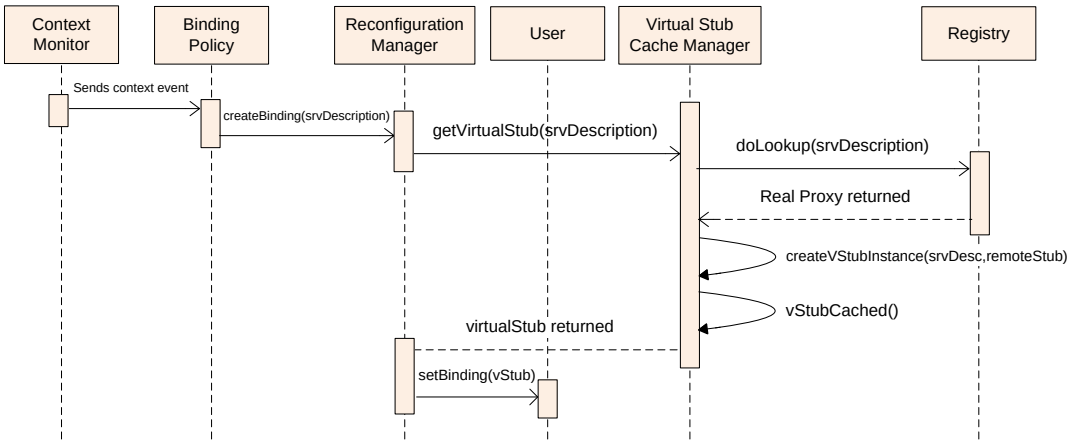

Figure 6.9: Reconfiguration time without cache sequence diagram

One of the features of the PCRA (discussed in chapter 3) is a provision of seamless caching support of virtual stubs for improved performance where, as a part of binding process, the virtual stub is obtained from a local cache if available for improved performance. To study the performance of this feature, we measure the reconfiguration time without using the cache (the remote service is discovered through a remote call to the registry and a virtual stub instance for this service is created and initialized with the corresponding real proxy and given to a user instance) and using cache (the virtual stub for the required service is obtained from the local cache directly without the need for a remote call and given to the user instance). Figure 6.10 shows sequence diagram for reconfiguration time with cache.

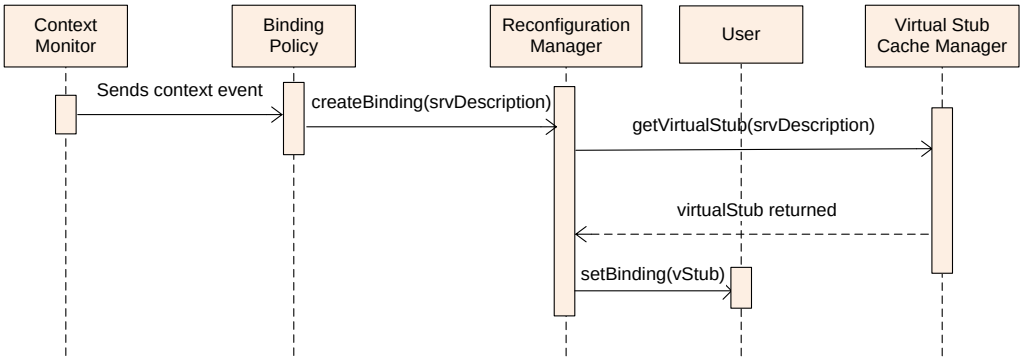

Figure 6.10: Reconfiguration time with cache sequence diagram

The adaptation time is measured from the point context is sent by the context monitor to the policy, which has subscribed to it, until the behavior of the service is adapted (by invoking a method with a parameter on the service) in response to policy evaluation. Figure 6.11 shows sequence diagram for adaptation time.

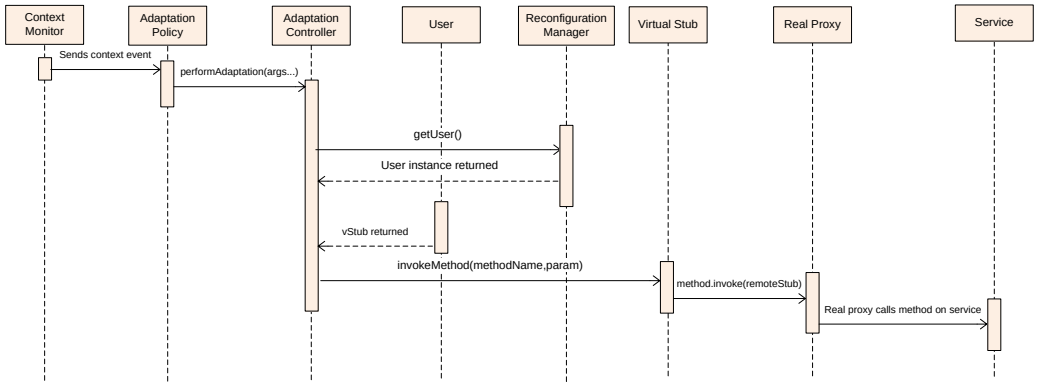

Figure 6.11: Adaptation time sequence diagram

In order to study how PCRA performs in a local setting and distributed setting, we have measured both reconfiguration time (with and without caches) and adaptation time under both settings.

6.3.1 Test environment

We conducted tests under both a local setting and a distributed setting. In the local setting we used four machines and ran tests on each of them. The four machines used were: (1) Intel Celeron 501 MHz, 256MB RAM; (2) Intel Pentium 3 734 MHz CPU, 384MB RAM; (3) Intel P4 2.4GHz, 1GB RAM; and (4) Intel Xeon, 2 GHz, 2GB RAM. All machines were running windows XP and used JDK1.5. The JDK1.5 or above is the requirement for running PCRA. As can be noted, machines used range from a less powerful machine to a typical desktop PC. The purpose of running tests on each of these machines was to investigate how PCRA scales from less powerful machines to powerful desktops. In the local setting, PCRA (Ponder2 system and reconfiguration and adaptation infrastructure), a RMI registry, remote service (LightService) and context monitors involved were running on the same machine. The local setup configuration is shown in figure 6.12.

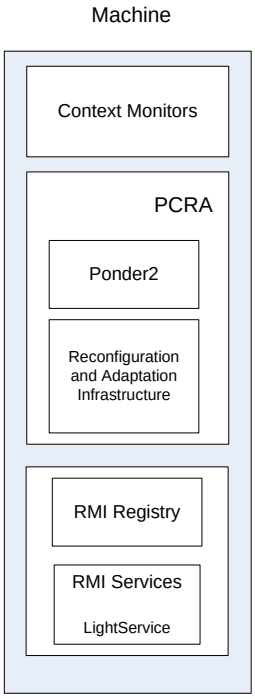

Figure 6.12: PCRA local setting configuration

In a distributed setting, two machines of the same specifications were used. These machines had the same specifications as that of one (Intel Xeon, 2 GHz, 2GB RAM) used in the local setting, and were connected through a 1 Gigabit wired network of school of informatics at University of Sussex. PCRA (Ponder2 and reconfiguration and adaptation infrastructure) and context monitors were running on one machine, while a RMI registry and remote service (LightService) were running on other machine. The distributed setup configuration is shown in figure 6.13.

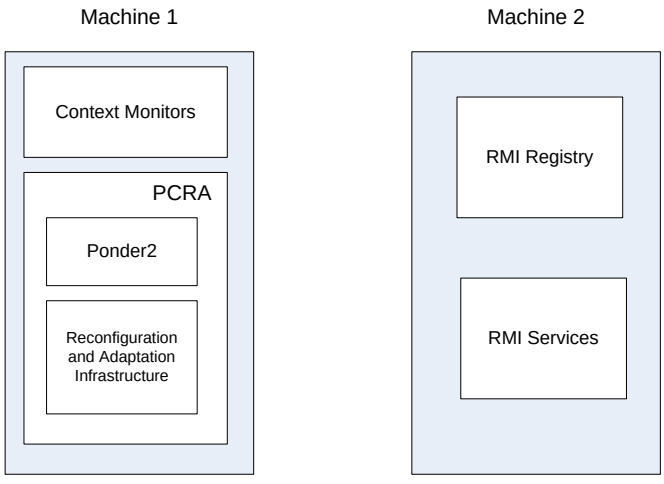

Figure 6.13: PCRA distributed setting configuration

6.3.2 Test Results

Below we present the results obtained for various performance tests under the local setting and distributed setting.

6.3.2.1 Local Setting

Test 1: Reconfiguration time without cache

To measure this time on each of four machines, the binding policy was triggered 20 times. This policy responds to a user presence event and its action part has a reconfiguration message to discover and bind the light service to the user instance.

Test 2: Reconfiguration time with cache

To measure this time on each of four machines, the same binding policy used in Test1 was triggered 20 times. In the binding process, the virtual stub was obtained from the local cache directly without the need for a remote call and delivered to the user instance.

Results: The reported times for test 1 and test 2 are average times and are presented graphically in figure 6.11 along with standard deviation. Reconfiguration time includes a RMI lookup time (time to discover a remote service) and this provides the largest contribution to reconfiguration time. This is due to fact that the remote calls are much slower than local calls, at least 1000 times slower. As can be noted, reconfiguration time with caches on each of machines is a far lower than the reconfiguration time without caches. This clearly shows that the seamless caching support of virtual stubs provided by PCRA significantly reduces reconfiguration time, hence improves performance. Note that the binding policy triggered in test 1 involves discovering only one remote service, light service (i.e., the user has a binding with one service), thus reconfiguration time includes just one remote lookup time. However, the user can have multiple bindings and each binding involves a remote lookup time. As a result, the reconfiguration time with each additional binding would increase significantly. As expected, as shown in figure 6.14, reconfiguration time with and without caches from a less powerful machine to a powerful machine reduces considerably and vice versa from a powerful machine to a less powerful machine. This indicates that PCRA can scale down and run on resource-constrained devices.

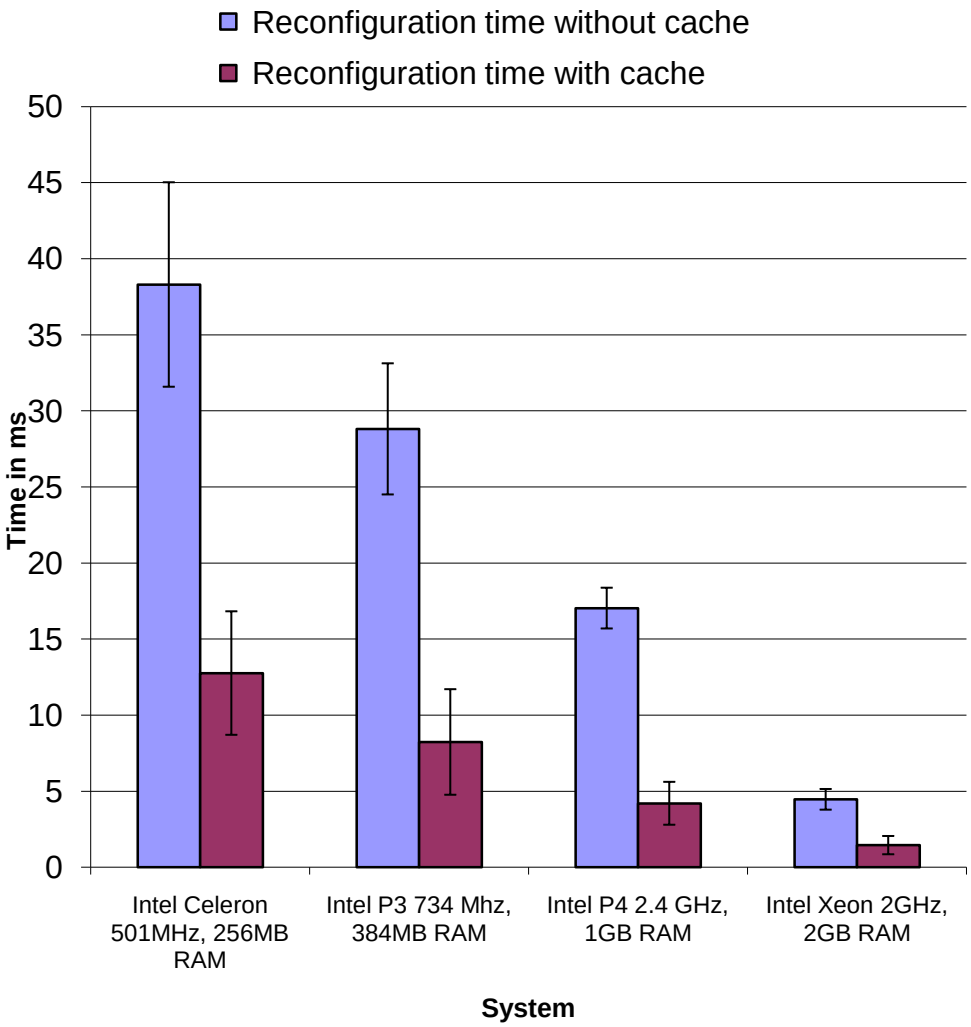

Figure 6.14: Reconfiguration time in local setting

Test 3: Adaptation time

In order to measure the adaptation time on each system, the adaptation policy was triggered 20 times. This policy responds to an activity context event and adapts the behaviour of a light service through a light value parameter adjustment if the condition is true.

Results: The reported adaptation time on each of the machines is an average time. Adaptation times on each of the machines are presented graphically in figure 6.15 along with standard deviation.

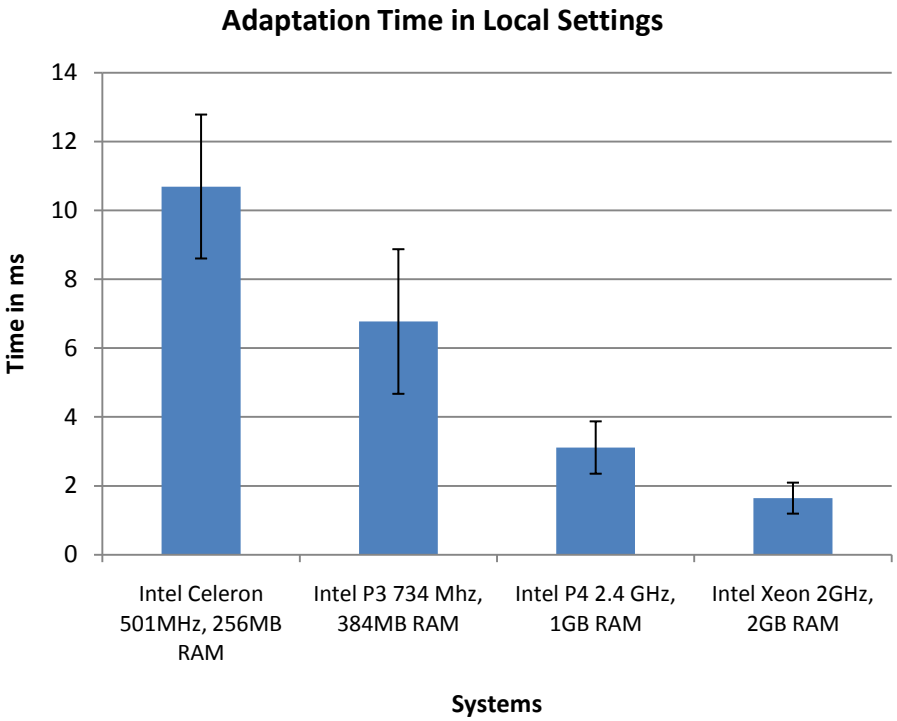

Figure 6.15: Adaptation time in local setting

6.3.2.2 Distributed Setting

We conducted the same three tests in a distributed setting. The purpose of running the tests in a distributed setting was to study the impact of the network.

Results: The results of our tests are shown and compared with the results in the local setting in a comparative analysis graph in figure 6.16. The reconfiguration time with and without caches, and adaptation time in distributed settings is a slightly higher than in the local setting. This indicates that PCRA performs reasonably well in the distributed setting; hence it can be deployed in the distributed setting.

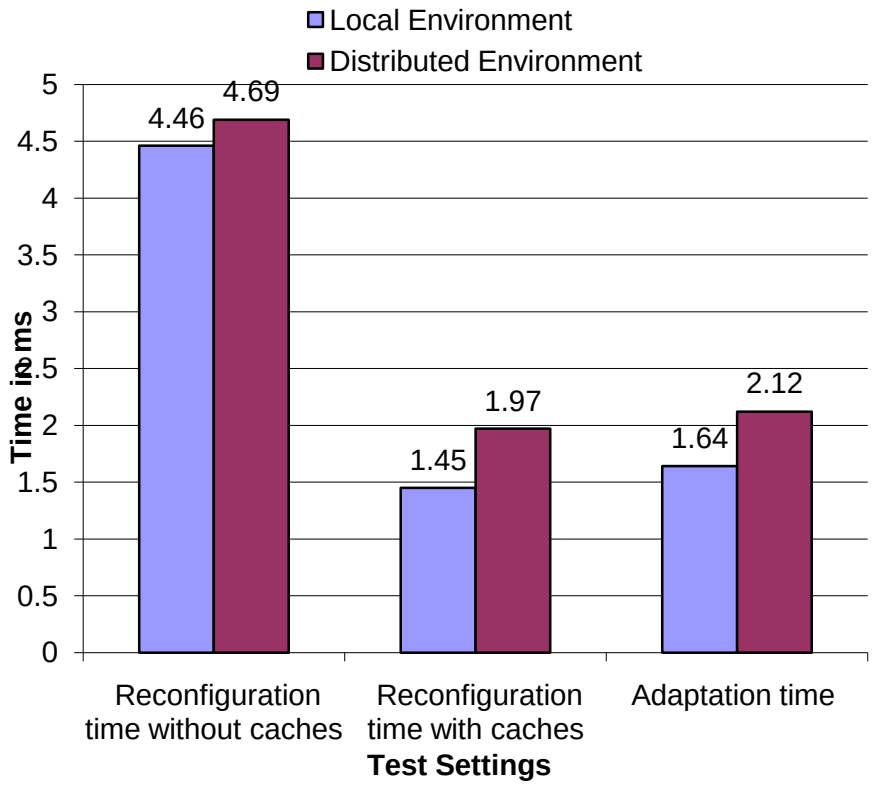

Figure 6.16: Comparative analysis of reconfiguration and adaptation time between local distributed settings

6.4 Summary

In our high-level analysis and qualitative evaluation sections, we conducted various tests to evaluate our proposition to decide if taking a policy-based programming approach for developing, modifying and extending adaptive context-aware applications is more effective than a specifically designed service composition language (Scooby) and the use of an API approach (One.World). We considered the simplification of development, dynamic modification, dynamic extensibility and the support for user involvement as important different aspects of effectiveness of an approach. We evaluated the effectiveness of our approach and compared against Scooby and One.World with regards to these aspects. We summarize the results of our evaluation in table 6.5.

Aspects of Effectiveness	**PCRA**	**Scooby**	**One.World**
Reduced Development Efforts	√	√	X
Dynamic Modification	√	X	X
Dynamic Extensibility	√	X	X
Support for User Involvement	√	Partially	X

Table 6.5: Effectiveness of approach comparisons

Performance results show that caching support of virtual stubs provided by PCRA significantly reduces reconfiguration time and improves system scalability. The overhead of distribution is small compared to both reconfiguration and adaptation time in local setting.

Chapter 7

Related Work

This chapter provides an overview of various systems that focus on providing adaptation support in ubiquitous computing environments. We discuss core concepts involved in each of the systems, adaptation types (i.e. reconfiguration, parameter adaptation, dynamic association and disassociation of non-functional concerns/low-level services, code mobility) supported and which approach is provided by these systems to develop adaptive context-aware applications. The approaches proposed in the literature to developing adaptive context-aware applications include API-based, specifically designed languages and policy-based. We group adaptation systems according to type of language used to develop adaptive context-aware applications and discuss how PCRA differs from these systems with respect to adaptation scope supported, application domains targeted and the approach to developing adaptive context-aware applications. This chapter also reviews some of the approaches used in the literature to handle the issue of updating invalid references in response to migration or replacement of the component, and compares them with PCRA's approach to handling this issue.

7.1 Approaches to developing adaptive context-aware applications

7.1.1 API-based approach

In this section we review various adaptation systems that provide an API-based approach to developing adaptive context-aware applications.

7.1.1.1 Enactor model

Dey et al. [21,22] have described systems that support adaptation of application parameters in response to context events. Their system provides a Java-based environment to build context-aware applications. In [22], Newberger and Dey have extended context toolkit [21] by introducing an enactor component to provide monitoring and control of context-aware applications. The idea behind this component is to encapsulate state information and adaptation logic of an application, and to allow external access. They have developed a Java API for

developing an enactor component and externally accessing application logic encapsulated in the enactor. Monitoring and control interfaces allow connection to the enactor via this API and accessing application logic to monitor and control context-aware applications. The enactor has three subcomponents: *references*, *parameters* and *listeners*. An enactor receives context input from context sources (context widgets) through *references*. *Listeners* facilitate monitoring, where they are notified of occurrences within the enactor, such as any actions that are invoked, contextual data received through *references* and *parameters* changed. *Listeners* send these occurrences to the monitoring and control interface where the interface changes the corresponding visual elements to reflect these occurrences. *Parameters* are various properties of the context-aware applications. For example, in a home lighting scenario, a light parameter is an integer value that specifies the light level to which light intensity in the room should be set upon the entry of the user to the room. The enactor exposes *parameters* and the user can control context-aware applications by manipulating these *parameters* via a monitor and control interface.

In [22], Newberger and Dey have described various example scenarios of domestic environments to demonstrate the usefulness of their enactor model approach for developing context-aware applications. We take one of their demonstration applications and highlight differences between our approach and theirs. The scenario is described below.

"when a person enters a dark room, its light is initially raised to 75% intensity, and dark adjoining rooms are raised to 10% intensity; if the person remains in the room for a certain time period, lights in the adjoining rooms fade slowly back to darkness".

To implement this functionality one enactor is required. It exposes a number of parameters, such as an integer specifying the light value to which light intensity is raised upon user presence, a Boolean type parameter specifying whether or not to turn on the light in adjoining rooms upon user presence, and the time after which adjoining rooms should go dark. Upon the user presence in the room, the enactor raises the light intensity of the light in the room to 75% intensity and the light in adjoining rooms are turned on and set to 10% intensity, and if the user stays in the room for a certain time period, lights in the adjoining rooms slowly go back to darkness. This behaviour can be controlled or modified by changing the required parameter values. For example, the light intensity of the light in the room upon user entrance can be raised to a value other than 75% intensity by modifying the parameter that specifies the light intensity. The behaviour of the application that lights in adjoining rooms are turned on upon user presence can be modified so that light in adjoining rooms are not turned on, by modifying the parameter of Boolean type (specifying whether or not to turn on the light in adjoining rooms) to false. Similarly, the parameter (specifying the time after which unoccupied room should go dark) can

be changed to modify the behaviour accordingly. Within PCRA, we have a user component that corresponds to an enactor component, which contains various parameters, each for a particular service (e.g. the light service, music service). The users can control or modify the behaviour of context-aware applications by manipulating these parameters using our GUI-based system utility (discussed in chapter 5). However, PCRA does not support monitoring of context-aware applications.

The point worth noting is that an enactor only exposes *parameters* to control or modify context-aware applications. However, there may come times when modification or control of application behaviour may involve more than just setting the parameters. For example, in the scenario above, it is desirable that the application behaviour may be extended so that when the user enters the room, in addition to the existing behaviour, the air-conditioning unit is also turned on and its setting is adjusted to its last value used. In order to achieve this using enactor model, the application logic that will perform this behaviour has to be added in the enactor. For this, the source code of the enactor needs to be modified, recompiled and restarted. This highlights two limitations of enactor model: (1) it affects user involvement in modifying or controlling the behaviour of the context-aware application in that the enactor is coded in Java and the user cannot be expected to be a Java developer, and (2) it does not allow dynamic modification of context-aware applications. In contrast, within PCRA, the application logic is specified declaratively at a high-level of abstractions as a rule-based policy, which enables end users to participate in developing and modifying context-aware applications. This improves user involvement in controlling or modifying context-aware applications. Moreover, polices can be loaded and unloaded dynamically without restarting the system. This provides the support for dynamic modification of context-aware applications.

7.1.1.2 Odyssey

Noble et al. [16,17,81] have developed the Odyssey system, which provides application-aware adaptation support for mobile applications in response to resource variation. Odyssey is an extension to the operating system, which provides a set of APIs to support adaptation for mobile applications. Odyssey runs on the client and the service (e.g. video server) runs on the server, and the client is connected with the service through a wireless link. Adaptation is achieved by modifying the service (e.g. dynamically selecting a version of content based on resource availability through parameter changes).

In Odyssey, the adaptation responsibility is shared between both the Odyssey system and the applications, where the Odyssey system is responsible for monitoring the resources and notifying the applications when the levels of these resources do not satisfy the resource

requirements. On notification, the applications choose a fidelity level (i.e. quality level) and send Odyssey the resource requirements for the chosen fidelity level. The fidelity level may be in the form of choosing a particular version of content out of multiple versions of the same content (e.g. black and white content for the movie, full colour with lossy JPEG compression, full colour uncompressed, etc.) or instructing the service through parameters to change the fidelity level (e.g. reducing frame rate, reducing the resolution of an image, etc.).

Adaptation support is provided by Odyssey through a *viceroy* and *wardens*. The viceroy is responsible for monitoring the resources and acts as a single point for clients' interactions with different *wardens*, while the *warden* is in charge of performing adaptations. The communication between the client and the *warden* takes place through the *viceroy*. Odyssey supports media-specific adaptation (e.g. video, audio), so it requires a specialized *warden* for each media type (e.g. a video *warden* for the video media type and an audio *warden* for an audio media type). Odyssey supports this media-specific adaptation through a general mechanism called "type-specific operation", or *tsop*. On a resource notification from the viceroy, the application selects a fidelity appropriate to the new resource variation via a *tsop()* call. Figure 7.1 shows the syntax of a *tsop()* call.

```
tsop(in path, in opcode, in insize, in inbuf, inout outsize, out outbuf)
```

Figure 7.1: Type-specific operation in Odyssey

The *path* argument to a *tsop()* call specifies the path of the corresponding *warden* and the *opcode* argument is the type of operation to be performed by the *warden*. This *tsop()* call is intercepted by the viceroy and invokes the *warden*, and then the *warden* performs the adaptation. Similar to this, we have three variants of adaptation message (Figure 7.2) in PCRA (discussed in chapter 5), which make up the adaptation interface of PCRA through which PCRA performs contextual-triggered actions and contextual adaptation by modifying these actions through parameter adjustments.

```
1. adaptation performAdaptation:#("serName" "loc" "methName" )
2. adaptation performAdaptation:#("serName" "loc" "methName" para )
3. adaptation performAdaptation:#("serName" "loc" "methName"   "userPreferredValue")
```

Figure 7.2: Adaptation interface in PCRA

As discussed above, on notification from Odyssey about resource availability, the application selects the fidelity appropriate to the new resource situation via a *tsop()* call. This means that applications implement adaptation policies which are hard-coded in the application. The hard-coding of adaptation policies in the application introduces inflexibility in that (1) it increases the complexity involved in developing applications because adaptation concerns are mixed up with the application functionality and (2) the adaptation policy of applications cannot be modified at runtime. This limitation can be addressed by following a separation of concerns principle in which adaptation policies would be specified separately from the coding of the application functionality. An interaction between the application and Odyssey would then take place through the adaptation policy (which is a separate and external component) which would receive notification from Odyssey and select a fidelity appropriate to the new resource variation via a *tsop()* call. With this, (1) legacy applications (e.g. video and audio players, browsers) can take advantage of application-aware adaptation support of Odyssey without having to modify their source code; (2) adaptation policies can be modified dynamically, without recompiling and restarting the Odyssey client and (3) the complexity involved in developing and maintaining adaptive applications is reduced.

Our work has similarities with Odyssey in that the part of overall adaptation support provided by PCRA involves modifying the service behaviour through parameter changes in response to context. Similarly, in Odyssey, adaptation involves modifying the service behaviour through parameter changes (e.g., selecting different versions of the same content through the change of parameter). The Odyssey adaptation support is targeted in mobile environments and supports media-specific adaptation (e.g. video and audio) only, where the media-specific service is modified in response to a change in resource availability. However, we target a set of application scenarios in domestic environments, where adaptation is performed in response to other contextual triggers such as environmental context (light level, noise level, temperature level) and user context (user presence, user activity, etc.). We believe that *wardens* can be implemented for adapting all types of services in response to contexts other than resource variation. The main difference between PCRA and Odyssey with regard to contextual adaptation support is that adaptations, within PCRA, are controlled through high-level declarative policies, while, within Odyssey, adaptation policies are hard-coded within application code.

7.1.1.3 One.World

Grimm et al. [76-78,84] at the University of Washington have developed One.World, which provides a platform for building pervasive computing applications. It integrates a set of services, such as service discovery, Remote Event Processing (REP), migration and check pointing that help structure pervasive computing applications. It provides programmers with a Java API to build pervasive computing applications. One.World uses basic abstractions: tuples, components and environments. Tuples make a uniform data model in which all data, such as events and queries are tuples. Components define functionality, which import and export asynchronous event handlers. The Environment serves as a container for tuples, components and other environments, and each application has at least one environment in which it store tuples and in which its components are instantiated.

All communications in One.World take place through asynchronous events that are processed by event handlers. Applications are composed from components that exchange events through imported and exported event handlers. The REP service provides a communication model that allows sending events to event handlers. It supports both point-to-point communication and service discovery, including support of two binding mechanisms: *early binding* and *late binding*. Events are sent to event handlers through these bindings. REP supports these features through three simple operations: *export*, *send* and *resolve*. In order for event handlers to be accessible from remote nodes, they need to be exported under symbolic descriptors (these are tuples), and a client sends events by specifying symbolic receivers. This exporting is performed by an *export* operation, establishing a mapping between the symbolic descriptor and the actual event handler. In *early binding*, first the resource name is explicitly resolved to the actual event handler by a *resolve* operation and then the message is sent to the event handler by a *send* operation. As can be noted, the *resolve* operation performs an early binding discovery lookup in which it takes a query and returns a matching event handler. The *send* operation also supports *late binding* in which it combines query resolution and event routing into a single operation. In this *late binding* process, the discovery service resolves a query and locates event handler(s) and then routes the event to the located event handler(s). In *early binding*, the resource is to be discovered only once and then events are routed to this discovered resource. Hence this is suitable when an application is required to repeatedly send events to the same resource. However, at the same time, with *early binding*, the application needs to be prepared to rediscover the receiver if its computing context changes. In contrast, in *late binding*, the resource is discovered when the event is to be routed to the resource. *Late binding* introduces a performance cost for every event sent in that each time the event is sent;

discovery service first resolves a query and locates the resource(s), and then routes event to these resource(s). The advantage of *late binding* is that it is the most responsive to change.

Various demonstration applications have been developed by the authors on the top of One.World and these include a text and audio messaging system, a music sharing system, etc. Others have also developed adaptive applications on One.World, for example, the Scooby author has implemented various adaptive applications of domestic environments (see 4.1.2) on both Scooby and One.World to compare them. PCRA supports context-aware bindings in which bindings to remote services are established in response to context. Context triggers the binding operation and can be viewed as late bindings since these are resolved as context occurs. Once bindings are created, they are used to communicate with bound services and may not require binding to them each time communication is required. This is similar to early binding in One.World. However, PCRA also provides application-transparent support for reconfiguration to manage invalid bindings in that when a binding becomes invalid due to power failure on the host device or migration of the bound service to another other machine, the system performs discovery to obtain an up-to-date remote reference of the bound service and updates the invalid binding.

One of the fundamental design differences between PCRA and One.World is that PCRA advocates and uses a policy-based programming model in which policies are declaratively specified to develop adaptive context-aware applications. A policy is a higher level approach for building applications than that of an API-based approach used by One.World; hence PCRA considerably reduces the complexity involved in developing adaptive context-aware applications. Further, the policy-based programming approach allows dynamic modification of applications. PCRA uses both event and RPC (Remote Procedure Call) models for communications. One of the aspects involved in developing adaptive context-aware applications in PCRA is passing contextual information (real-world events) to a policy where policies react to contextual information to perform adaptations. The event model is a natural fit for this purpose in which contextual information is wrapped in an event and sent to policy that subscribes to it, while RPC is used by the policies to communicate with the bound service to modify its behaviour in response to context.

7.1.2 Specifically designed languages

In this section we review some of the related adaptation systems that advocate the use of specifically designed languages for developing adaptive context-aware applications.

7.1.2.1 RCSM

Yau et al. [24] have developed the middleware, Reconfigurable Context-Sensitive Middleware for Pervasive Computing (RCSM) that facilitates the development and runtime operations of context-aware applications in pervasive computing environments. In RCSM, the context-aware applications are modelled as context-sensitive objects, which consist of two parts: a context-sensitive interface and the object implementing the actions (functionality), which is context-independent. The context-sensitive interface is used to specify the context, its requirements and actions to be triggered, and the mapping between the specified context and these actions. RCSM provides a context-aware interface description language (CA-IDL) that allows the developer to specify a context-sensitive object interface. The CA-IDL compiler compiles the context-sensitive interface and generates an application-specific adaptive object container (ADC). The generated ADC is tailored for a particular context-sensitive object and includes support related to context specified in the context-sensitive interface. Context support included in the ADC is context data acquisition, monitoring and detection. At runtime, ADC communicates with the underlying system to obtain context data and also communicates with object implementation to invoke actions in response to context, performing context-triggered actions. RCSM also provides transparent support for ad hoc communication in which it discovers new devices and creates links between them.

The strength of the RCSM approach comes from the use of the CA-IDL language, which allows the application developer to specify the context requirements at a high-level of abstractions and the tasks related to context data acquisition and monitoring are performed by the underlying system. As can be noted from above discussion, RCSM maintains a clean separation between context-related operations (context data acquisition, monitoring and detection) and application functionality, where functionality is separately provided by a context independent application object. These features of RCSM reduce the complexity involved in developing and maintaining context-aware applications. With this approach, new types of context and context-aware behaviours can be incorporated by editing context-sensitive interfaces, making the modification of context-aware applications easier. However, this requires re-compiling and re-running the application. Similarly, the strength of PCRA comes from the use of a policy-based programming model which offers two main advantages: (1) adaptive context-aware applications are modelled as high-level declarative policies, thus reducing the complexity involved in developing such applications and (2) this allows dynamic modification of the applications in that the modification is made in the corresponding policy and then modified policy is dynamically loaded into system without interrupting the system.

7.1.2.2 Scooby

Robinson [23,69,95] has developed a domain specific composition language that provides a platform for developing pervasive computing applications. The development of pervasive computing applications in Scooby involves developing composed services using high-level language constructs. The composed service is compiled by the Scooby compiler, which produces the corresponding Java source code, and then this Java source code is compiled to produce Java byte code. The main idea of Scooby is that a dedicated domain specific language is a more efficient way of performing service composition than traditional approaches that use an API. One of the core features of the Scooby language is the concept of binding variables, which are high-level constructs and are used to develop composite services. The use of high-level language constructs for coding composite services makes their development easier than other traditional approaches where the composite services are written/developed using an API. Our work and Scooby share similar research goals in that we both advocate the use of a high-level means to achieve service composition/reconfiguration to simplify development, however, we use different approaches to achieving them. The fundamental design difference between the two is that PCRA uses a policy-based programming approach, where binding policies are specified declaratively at a high-level of abstraction in which reconfiguration messages (similar to binding constructs in Scooby) are expressed in the action part of the policy. In response to context, the policy is executed when its condition is met, causing reconfiguration messages in its action part to execute and perform reconfiguration.

In Scooby, binding variables are hard-coded in the composite service, thus introducing inflexibility in the sense that any change or modification in bindings requires modifying the source code of the composite service in question, recompiling and then restarting it. In contrast, in PCRA, reconfiguration messages are expressed in the policy and the policy is an explicit component which can be dynamically loaded and unloaded. Therefore, any changes in bindings require modifying the corresponding policy and reloading it dynamically without the need for shutting down and restarting the system.

The high-level Scooby constructs, such as binding variables have a direct mapping to the underlying processes available within the Scooby middleware. So, when the composed service is run and the binding variable executed, the corresponding thread is created, which is responsible for establishing a binding with a particular service and managing it throughout the life cycle of the binding. As a result, there would be as many threads executing as the number of binding variables involved in the composed service, each for managing the binding with a particular service. In Scooby, the binding can switch between various states (unresolved, discovery, connected, suspended and waiting) during its life time depending on whether a

remote service has been identified or not. For example, if the service is successfully discovered based on the search criteria and bound to, the binding is said to be in a connected state and allows calls to be made on the bound service. When a call is made on the bound service, the binding first checks if the bound service is still available by initiating contact with it and if available, the call is made; and if not, it enters the discovery state. Moreover, the binding periodically checks to make sure that the bound service is still reachable. If, for some reason, it is no longer available, the binding enters the discovery state. As can be noted from above discussion, Scooby provides a flexible binding model that is well suited to the pervasive computing settings, but at the cost of system performance and which may affect the user experience. This performance cost is due to the following factors—the concept of bindings switching in various states, a thread running for each of the bindings (periodically checking availability of the bound services and switching the binding state accordingly) and before making a call on the bound service, contacting the bound service to make sure if it is available. In contrast, PCRA provides a binding model in which the system discovers and binds the services based on contextual information and other search criteria as specified in reconfiguration messages. Unlike Scooby where a binding is periodically checked for invalidity, and also prior to a remote method call, in PCRA, when a remote method call is made on the bound service and the binding has become invalid, an exception is thrown. In response to this exception, the system performs reconfiguration to update invalid binding and then repeats the method call. In addition, PCRA also provides caching support of virtual stubs (discussed in chapter 3) for improved performance.

7.1.3 Policy-based approach

This section reviews various adaptation systems that advocate the use of policy-based programming model to develop adaptive context-aware applications.

7.1.3.1 Towards a framework for self-adaptive component-based applications

David and Ledoux [12] have presented a framework, which enables the development of self-adaptive component-based applications. Their framework supports two adaptation mechanisms: reconfiguration and parameter adaptation. The motivation behind this framework is to enable systematic development of adaptive mobile applications using a “separation of concerns” principle, where adaptation concerns are separately treated and defined from the core functionality of an application. The adaptation concerns of the application are defined in an

ECA (Event-Condition-Action) format, called adaptation policies. This separation of concerns reduces the complexity involved in developing adaptive applications. Further, adaptation policies can be modified during execution, without stopping and restarting the system, thereby allowing dynamic modification of applications.

This framework uses policies for controlling both kinds of adaptation mechanisms (reconfiguration and parameter adaptation), in which adaptation policies respond to events (representing contextual information) and trigger reconfiguration actions. Reconfiguration actions may involve the reconfiguration of application architecture or adaptation of the behaviour of the component through parameter adjustments. Adaptation polices are currently specified using the Java language. PCRA has a close resemblance with this framework in the sense that it also supports reconfiguration and parameter adaptation, and a policy-based approach for controlling adaptations. However, the reconfiguration supported by PCRA is different in the sense that the system discovers the services based on context (e.g. user presence) and creates bindings between the user component and these services, unlike the structural reconfiguration supported by this framework where configurations (which components to bind) are statically defined. In addition, PCRA also provides reconfiguration support to manage invalid bindings, thus offering a broader scope of adaptation than this framework. Our approach also differs from this approach with regard to how the adaptation policies are specified. This approach currently uses the Java language and plans to develop a domain specific language as future work to encode adaptation policies, while our approach uses a specialized declarative policy language, Ponder2 [26,80,94], to specify adaptation policies. The advantage of using Ponder2 for specifying adaptation policies is that policies are specified declaratively at high-level of abstraction, thereby simplifying development.

7.1.3.2 SCaLaDE

Bellavista et al. [10] have developed the Services with Context-awareness and Location-awareness for Data Environments (SCaLaDE) middleware that focuses on mobile environments and provides the support for reconfiguration of links to information resources in response to the mobile entity migration. In addition, it also provides adaptation support where service results are adapted to fit specific device characteristics; and also handles temporary disconnections during which service requests are carried on. SCaLaDE creates a mobile proxy (also called a shadow proxy) for each mobile device. It is implemented as a mobile agent with bindings to information resources. The mobile agent follows the mobile client movements and updates its bindings with information resources as it migrates due to roaming of mobile clients.

In SCalaDE, a mobile proxy can refer to information resources through various types of binding strategy: resource movement, copy movement, remote reference and rebind [54]. The decision of which binding strategy should be used between a mobile proxy and the required resources when the mobile terminal/mobile proxy moves is based on deployment conditions, such as terminal capabilities (e.g. CPU power, memory space etc.) and available bandwidth etc. These binding strategy decisions are expressed explicitly through high-level policy specifications, thus providing a separation of concerns between binding management concerns and application logic. This separation of concerns reduces the complexity involved in development of mobility-enabled scenarios and allows dynamic programmability of binding strategies. SCaLaDE expresses the binding strategies as Ponder obligation policies [27]. The use of a policy-based approach to separate binding management concerns from the computational aspects is the core idea of SCaLaDE.

As can be observed from the above discussion, context-awareness here captures the binding management concerns where, in response to context (migration of a mobile component), a particular binding strategy is selected based on deployment conditions and then the link between the mobile proxy and the needed resource is reconfigured accordingly. Context-awareness is controlled through a policy specification. Although PCRA supports other forms of context-awareness: contextual reconfiguration (services are discovered and bound to in response to context) and contextual adaptation, both PCRA and SCaLaDE advocate and use a policy-based programming model to reduce complexity involved in developing context-aware applications. However, PCRA uses Ponder2 for expressing obligation policies, while SCaLaDE uses Ponder obligation policies.

7.1.3.3 POEMA

Montanari et al. [66] have developed a policy-based framework, Policy-Enabled Mobile Applications (POEMA) that provides an environment to develop applications that can change their functionality as well as layout dynamically in response to context (e.g. user mobility, low battery power, etc.). POEMA exploits code mobility which enables dynamic change of the application deployment by transferring execution of software components from one device to another depending on resource availability.

The core argument presented in this work is that mobility complicates development efforts in the sense that developers are required to define when and where to move which components under varying operating conditions. As a solution to this problem, a policy-based programming approach has been advocated and used, where the POEMA platform supports high-level reconfiguration strategies expressed as ECA rules that separate mobility concerns

from application functionality. Policy specifications express a choice regarding when, where and which components to move in response to context. Developing mobile code applications under POEMA requires expressing mobile concerns separately from the coding application functionality. This separation reduces the complexity involved in developing mobile-code applications, and also allows dynamic modification of mobility concerns, thus allowing dynamic reconfiguration of mobile-code applications.

PCRA also advocates and uses a policy-based programming approach for reconfiguration to simplify the development task and to allow dynamic modification of the applications. However, the reconfiguration used in PCRA is different from POEMA in that reconfiguration involves discovering service(s) and binding to them in response to context. In addition, PCRA also supports policy-based contextual adaptation in which the service behaviour is adapted through parameter adjustment in response to context. PCRA uses Ponder2 for specifying binding and adaptation policies, while POEMA uses Ponder for expressing reconfiguration strategies.

7.1.3.4 Chisel

Keeney and Cahill [13] have developed a policy-driven, context-aware, dynamic adaptation framework, Chisel, which facilitates the development of context-aware applications for mobile environments. Chisel uses dynamic association and disassociation of non-functional concerns with the core functional concerns as an adaptation mechanism. This framework is based on the idea that particular aspects of the service object which do not provide core functionality (non-functional concerns) should be separated out into multiple possible behaviours. The adaptation of service objects is then achieved by associating non-functional concerns with and disassociating from them at runtime in response to context, without stopping their execution.

Chisel uses a policy-based programming approach in which policies are specified to control adaptations. They have developed their own policy-language, which includes constructs like ON-DO-IF. Policy specifications maintain a clean separation between the decision logic that determines when to perform what adaptations and actual adaptation actions, thus facilitating easier and rapid development and maintenance of adaptive applications.

Our work has a resemblance in the sense that we also use policies for controlling adaptations, but we focus on other adaptation mechanisms—dynamic reconfiguration of an application, which creates bindings between application components in response to context, and the parameter adaptation to achieve contextual adaptation.

7.1.3.5 CASA

Mukhija [1,82,83] has developed a Java-based framework, called CASA (Contract-based Adaptive Software Architecture), which enables the development of dynamically adaptive applications. The core idea of this framework is a provision of a wider scope of adaptation with an aim to comprehensively meet the adaptation needs of applications running in dynamic environments. The broader scope of adaptation is provided by integrating a number of different adaptation mechanisms, i.e. dynamic reconfiguration of application components, dynamic association and disassociation of non-functional concerns, dynamic changes of lower-level services and dynamic changes of application attributes. CASA considers both dynamic changes of lower-level services and dynamic changes of application attributes as separate adaptation mechanisms. However, we argue that both can be achieved through parameter adjustments and therefore can be considered to be a parameter adaptation mechanism. In the following, we refer to these two as a parameter adaptation mechanism.

The authors [1,82,83] have developed their own approach for dynamic reconfiguration of components in the CASA framework. For providing dynamic association and disassociation of non-functional concerns adaptation support, a third party system called PROSE [56] has been integrated in the CASA framework. Similarly, for providing the support for dynamic changes of lower-level services, another third party system, Odyssey has been integrated in the CASA framework. Having integrated these two third party systems and their approach for dynamic recomposition of application components, the CASA framework provides a wide scope of adaptation.

In CASA, an adaptation policy of an application is defined in XML that is external to the application. This provides a separation of concerns between adaptation concerns and core application functionality. As a result, two major benefits are achieved: (1) it reduces the complexity involved in developing and maintaining dynamically adaptive applications and (2) it allows modification of the adaptation policy at runtime.

PCRA also provides a broader scope of adaptation by integrating a number of adaptation mechanisms: reconfiguration, parameter adaptation and reconfiguration to manage invalid bindings. Two adaptation mechanisms (application code reconfiguration and parameter adaptation) supported by both the systems are the same. However, the third adaptation mechanism supported by CASA is a dynamic association and disassociation of non-functional concerns, while PCRA supports reconfiguration to recover from invalid bindings. In addition, PCRA also integrates seamless caching support of virtual stubs for improved performance. As can be noted from above discussion, CASA uses third party systems for supporting adaptation

of services and other adaptation involving association and disassociation of non-functional concerns, while we have developed our own approaches for all kinds of adaptation supported by PCRA. In contrast to CASA where adaptation policies are specified in XML, PCRA uses a specialized declarative policy language, Ponder2, for encoding adaptation policies. In earlier versions of Ponder2, XML was used for writing policies but later it was argued that writing XML was laborious for human and also hard to debug and read. As a result, high-level language called PonderTalk (see section 2.6.2.3) was developed for expressing Ponder2 policies.

As discussed above, CASA integrates Odyssey system for providing support for adaptation of lower-level services and its adaptation support is limited to media-specific adaptation (e.g., video and audio) in mobile environments, where media-specific service is adapted in response to resource availability. In contrast, we target a set of application scenarios in domestic environments; where service (e.g., light service, cooker service, fridge service, etc) is adapted in response to other contextual triggers such as user presence, user activity, light level, noise level, temperature level, etc.)

7.1.4 Summary

In above sections, we discussed core concepts involved, adaptation scope supported, and the programming approach provided by each reviewed system to develop adaptive context-aware applications. We summarize our observations in following two tables, where table 7.1 shows adaptation scope supported by each reviewed system, while table 7.2 shows which programming approach is provided by each of them.

Systems	Adaptation Scope				
	Reconfiguration/ Binding	Parametric	Dynamic Association & Disassociation of Non-Functional Concerns	Code Mobility	Reconfiguration to update Bindings
Enactor model		√			
TFSACBA[7]	√	√			
Odyssey		√			
CASA	√	√	√		
One.World	√				√
Scooby	√				√
SCaLaDE	√			√	√
POEMA				√	√
Chisel			√		
RCSM	√				
PCRA	√	√			√

Table 7.1: Adaptation scope comparisons

[7]Towards a Framework for Self-Adaptive Component-Based Applications.

Systems	API	Programming Language	Policy
Enactor model	√		
TFSACBA[8]			√
Odyssey	√		
CASA[9]			√
One.World	√		
Scooby		√	
SCaLaDE[10]			√
POEMA[11]			√
Chisel[12]			√
RCSM		√	
PCRA[13]			√

Table7.2: Programming approach comparisons

7.2 Approaches to updating invalid references

In [61], the authors identify various failure conditions and robustness issues that can arise in context-aware pervasive computing applications and provide application-level recovery mechanisms. However, we only focus on binding failures that affect an interaction between an application and the bound service. The binding failures can be caused by either power-failure problems, or migration or replacement of the bound service. Although they handle various failure conditions and hence their solution is comprehensive, their solution is not application transparent. In contrast, our reconfiguration approach to managing bindings is application transparent.

Most of the current mobile frameworks use a system level approach to maintain a reference with a moving target object and they normally solve this problem with one of the two approaches. The first approach is to continuously maintain a valid reference to the moving target using a tracking mechanism as done in [62,63]. The tracker is a forwarding pointer. Upon

[8] TFSACBA uses the Java programming language to code adaptation policies.

[9] CASA uses XML language to encode adaptation policies.

[10]SCaLaDE uses Ponder to express adaptation policies.

[11] POEMA uses Ponder to express adaptation policies.

[12] Chisel uses their own developed policy language to express adaptation policies.

[13] PCRA uses Ponder2 to express adaptation policies.

the migration of a component to other location, the system creates a tracker in the old place. The method calls are forwarded to the moved target object by that tracker. While this approach provides an application transparent mechanism to always maintain a valid reference to the moved component, it is costly in two respects: (1) the system must create a tracker in the old place and (2) the method calls are first received by the tracker and then forwarded to the moved object. The second approach is to rebind the reference to the relocated object each time a method of the target object is called so that the reference to the moved target is always valid. This approach is very costly from the system performance point of view because each time the target method is called, a lookup operation is performed and this is very time consuming. In contrast, our approach does not suffer from these system performance issues.

Other systems that provide application transparent support for managing references upon component migration or replacement include [47,64]. These systems use a system design component called a virtual stub/smart proxy that holds/wraps the real proxy of the target component. An interaction between the application component and target object takes place through the virtual stub/smart proxy. In an attempt to call a method on an invalid reference, the virtual stub/smart proxy performs reconfiguration to update the invalid reference, thus providing an application transparent reconfiguration. In [64], the smart proxy catches an exception generated by an attempt to invoke a method on a target object which has since been replaced, and attempts to update the reference by looking up the replacement object in a naming service. In [47], a virtual stub updates the invalid reference when asked by the system or in response to an exception, generated by an attempt to invoke a method call on the target object which has since been moved or replaced. Our approach is similar to theirs in that our approach also uses a virtual stub and thus it is application transparent, but it is different in that the virtual stub reflectively invokes remote methods of the bound service. This allows having a generic virtual stub definition which can be used to wrap a real proxy to any remote service without knowing a remote interface implemented by the remote service. This means the same virtual stub definition is used for each unique remote service without the need for having the definition of each remote interface implemented by a remote service available locally. This results in a reduction in total amount of code.

7.3 Summary

In this chapter, we provided an overview of various systems that focus on providing dynamic adaptation support in pervasive computing environments. We discussed the core concepts involved, adaptation mechanisms used by these systems to realize dynamic adaptation, and also their approaches to developing adaptive context-aware applications. Adaptation

mechanisms used by these systems to achieve dynamic adaptation were among the ones we identified from the literature in chapter 2 (dynamic reconfiguration of application components, parameter adaptation, dynamic association and disassociation of non-functional concerns, code mobility).

As discussed in chapter 2, each of the adaptation mechanisms enables powerful adaptations. Integration of different adaptation mechanisms provides a broader scope of adaptation to meet the diverse adaptation requirement of the application. Table 7.1 summarises adaptation scope provided by all reviewed systems by integrating various adaptation mechanisms. One of the arguments of this thesis is the provision of a broader scope of adaptation to meet different adaptation needs of applications. Table 7.1 indicates that PCRA meets this argument. Odyssey provides parameter adaptation support only and target adaptation of media-specific applications in response to resource variability, while PCRA integrates other adaptation mechanisms and targets application scenarios in domestic environments. Both SCaLaDE and POEMA use code mobility to enhance service provisioning to mobile users and to address the limitation of resource-constrained devices, and both targets mobile environments. However, SCaLaDE provides a wider adaptation scope than POEMA as it also includes reconfiguration support for location-dependent services. In contrast to these two, PCRA does not use code mobility, but it integrates different adaptation mechanisms to provide a wider scope of adaptation. While Scooby, One.World and PCRA support reconfiguration for establishing bindings between software components, and reconfiguration support for managing invalid bindings, PCRA integrates an additional adaptation mechanism—parameter adaptation. Reconfiguration support in One.World involves discovering event handlers and binding to them, while in both Scooby and PCRA, it involves discovering services and binding to them. The other system that comes closer with regard to this is CASA, which also provides a broader scope of adaptation by integrating three adaptation mechanisms. Two of the adaptation mechanisms (application reconfiguration / binding and parameter adaptation) supported by both systems are the same. However, we address an issue of updating invalid references and provide reconfiguration support to handle this issue, while CASA does not address this issue, but includes dynamic association and disassociation of non-functional concerns adaptation mechanism.

The main motivation behind all of the reviewed systems was the provision of an approach targeted to reduce complexity involved in developing and maintaining context-aware applications. The approaches used by the systems (see table 7.2) were: a policy-based programming model, specifically designed language and an API-based approach. While all systems contributed to this goal, the systems that used the policy-based programming model

allowed dynamic modification of context-aware applications, without the need for recompiling and restarting the system. It can be argued that the policy-based programming model provides more effective means for developing, modifying and maintaining context-aware applications than specifically designed languages and an API-based approach.

Chapter 8

Conclusion

This thesis has provided a description and evaluation of our PCRA system, which enables the development and operation of adaptive context-aware applications. In this last chapter, we summarize our research and briefly describe some of the important research directions for future work.

8.1 Recapitulation

8.1.1 Motivation

Chalmers [5] suggests that context-awareness can manifest itself in a variety of ways. We are interested in the ways context-aware systems adapt themselves in response to context, in particular in contextual reconfiguration and contextual adaptation. These provide adaptations and enable applications to perform tasks which support us in everyday life with minimal user distractions. One of our motivations in this thesis has been to provide a broader scope of adaptation to meet the different adaptation needs of applications than existing systems. Each kind of adaptation to context can individually satisfy some adaptation requirements. However, we envision many adaptive context-aware applications, which require support for both reconfiguration and adaptation. Integrating support for both is therefore appropriate. Another motivation behind this research work has been to provide an effective means to simplify development of adaptive context-aware applications and to allow dynamic modification of such applications. Many existing research efforts have focused on this and provide means in the form of an API or the development of special purpose languages. However, we provide developers with a policy-based programming approach to simplify the development task and to provide support for dynamic modification of adaptive context-aware applications. Our evaluation results (chapter 6) showed that the policy-based programming approach provides simplification of development and supports dynamic modification of applications.

8.1.2 Summary of Contributions

The resulting effort of our research is the design and implementation of a system, PCRA, which provides a policy-based platform in which to develop adaptive context-aware applications. This acts as a proof of concept for the more general questions below:

- ***The provision for a broader scope of adaptation*:** PCRA supports three different forms of adaptation*: contextual reconfiguration*, *contextual adaptation* and *reconfiguration to manage invalid bindings*. Contextual reconfiguration allows context-aware bindings in which remote service(s) are discovered based on context and bound to user instances. Contextual adaptation employs a service parameter adaptation mechanism to achieve contextual adaptation in which the behaviour of a bound service is adapted through parameter adjustments in response to context. If a binding become invalid, reconfiguration support for managing bindings takes charge and updates the binding.
- ***Seamless caching of virtual stubs for improved performance*:** We have integrated seamless caching support of virtual stubs in PCRA for improved performance. As a part of our context-aware binding process, the remote service is discovered, the virtual stub created and initialized with the real proxy of the discovered service, and then cached locally. When a new binding to this service is required again, the corresponding virtual stub is obtained from the local cache directly without performing a remote look up. This considerably reduces reconfiguration time and hence improves the system responsiveness.
- ***Reducing the complexity involved in developing adaptive context-aware applications*:** Following detailed research, we decided to use a policy-based programming approach. PCRA is built on top of Ponder2 system, which provides support for specifying and enforcing policies. PCRA provides a policy-based programming platform to develop adaptive context-aware applications in which policies are expressed declaratively at a high-level of abstraction, thus reducing the complexity involved in developing such applications. PCRA demonstrates that policies provide an effective way to develop such applications.
- ***Runtime modification of applications*:** Development of adaptive context-aware applications under PCRA requires specifying both binding and adaptation policies which can be loaded and unloaded at runtime, allowing dynamic modification of applications during application execution.

8.1.3 Summary of Results

The two primary objectives of our research are (1) integration of various adaptation mechanisms to provide a broader scope of adaptation and (2) the provision of an effective means for developing adaptive context-aware applications. To justify these two main objectives of our thesis, we have implemented various hypothetical adaptive context-aware applications, and evaluated our policy-based programming approach and compared it with a specifically designed language (Scooby) and an API-based approach (One.World). We summarize our results below.

- The results of our literature survey show that PCRA provides a wider scope of adaptation by integrating more adaptation mechanisms than that of both Scooby and One.World.
- Our evaluation results show that both PCRA and Scooby reduce the complexity involved in developing adaptive context-aware applications as was indicated by the number of source code lines required to implement various example scenarios, while One.World requires significantly more development effort.
- We demonstrated that PCRA outperforms both Scooby and One.World by being able to make dynamic, rather than static, modifications to policies.
- We demonstrated that PCRA provides better user involvement support in specifying or reconfiguring adaptive context-aware applications than both Scooby and One.World. However, Scooby supports user involvement better than One.World.
- PCRA can handle complex scenarios well. This fact is highlighted by the implementation of various hypothetical example scenarios which represent real life home scenarios.
- Our performance evaluation results suggest that PCRA is scalable and can scale down to resource-constrained devices. This feature is inherited from Ponder2 which PCRA is built on top of.
- Performance evaluation results clearly indicate that caching support integrated within PCRA significantly reduces reconfiguration time and thus improves system performance.

8.2 Future Work

This thesis addresses research involving two interesting domains: context-awareness and policies. Consequently, there is a variety of interesting work which is beyond the scope of this thesis. In the following, we suggest some of the work which is worthy of further investigation.

User conflicts / Policy conflicts: The main focus of our research has been a provision for a broader scope of adaptation support, a policy-based programming approach to simplify the development process and to allow dynamic modification of adaptive context-aware applications. However, there are other important related issues which we suggest need to be addressed. Currently, we have embedded a very simple solution to the user conflict problem in PCRA, which is priority-based (the older the user, the higher the priority). This issue needs to be further explored and may require investigating various resolution strategies. The resolution strategy may be that an average of preferred values of multiple users for a particular service be taken and then checked if the average value is not far more deviated from each user's preferred value, averaged value be used. If the average value is too far deviated from each user's preferred value, then other strategies may be adopted (e.g., a priority-based scheme). The highest priority scheme may be based on the age or role of the user. In a role-based priority scheme, father or mother may get higher priority than their children, or in academic environment, faculty members may get higher priority than students. In priority-based schemes, priority of users might tie and require considering other factors (e.g., in shared accommodation, the user who has spent longer than others may get higher priority, etc.). Conflicting situations may also arise in policy-based systems when multiple policies sharing the same output are triggered and executed, leading to conflicting actions. In order to exploit the full potential of our policy-based approach, we suggest that policy conflict issues need to be investigated and its runtime support be integrated within PCRA.

Increased Generality of the system: PCRA is working and we have implemented various hypothetical adaptive context-aware example scenarios in domestic environments, and run them on it. From this we draw the conclusion that it offers reasonably good adaptation support in such scenarios. However, it is interesting to investigate whether its adaptation support can be exploited in other environments (e.g., mobile environments). We suggest that it should be tested in mobile environments and if any particular issues come up, these need to be addressed and incorporated to provide the necessary functionality. We further suggest that, in order to improve its generality, other adaptation mechanisms (e.g., dynamic association and disassociation of non-functional concerns) may be integrated within PCRA to broaden its adaptation scope so that it can satisfy diverse adaptation needs of applications.

Integration with existing context toolkits: The application scenarios implemented involve responding to various contexts (e.g., user presence, activity context, time etc.). Currently, we are using our own GUI components to generate simulated contexts to the applications. However, in order to deploy PCRA in real world scenarios, existing context toolkits (e.g., Dey's context toolkit [21]) can be used to provide contextual information from real world sensors. PCRA is

built on top of Ponder2 which uses an event model for receiving data from outside through events. All data sent from outside entities to PCRA must be a Ponder2 managed object. As a result of this, no existing context toolkits can be used directly with PCRA. Therefore, we suggest development of software that would enable to integrate PCRA with existing toolkits.

Bibliography

[1] A. Mukhija, "CASA - A Framework for Dynamic Adaptive Applications", Doctoral Thesis, University of Zurich, 2007.

[2] C. Efstratiou, "Coordinated Adaptation for Adaptive Context-aware Applications", PhD thesis, Lancaster University, United Kingdom, 2004.

[3] A. K. Dey and G. D. Abowd, "Towards a Better Understanding of Context and Context-Awareness", *Proceedings of Workshop on The What, Who, Where, When, and How of Context-Awareness, Conference on Human Factors in Computing Systems*, The Hague, The Netherlands, April 2000.

[4] N. B. Schilit, N. Adams, and R. Want, "Context-Aware Computing Applications", *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pp. 85-90, 1994.

[5] D. Chalmers, "Contextual Mediation to Support Ubiquitous Computing"**,** PhD thesis, Imperial College London, 2002.

[6] P. McKinley, S. Sadjadi, E. Kasten, and B. Cheng, "A Taxonomy of Compositional Adaptation", *Technical Report MSUCSE- 04-17*, Michigan State University, 2004.

[7] A. K. Dey, "Understanding and Using Context*", Personal and Ubiquitous Computing*, vol. 5, no. 1, pp. 4–7, 2001.

[8] J. Pascoe, "Adding Generic Contextual Capabilities to Wearable Computers", *Proceedings of 2nd International Symposium on Wearable Computers*, pp. 92-99, 1998.

[9] G. Chen and D. Kotz "A Survey of Context-Aware Mobile Computing Research" *Dartmouth College Technical Report TR2000-381*, November 2000.

[10] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli, "Policy-Driven Binding to Information Resources in Mobility-Enabled Scenarios", *Proceeding of 4th International Conference on Mobile Data Management*, 2003.

[11] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli, "Context-Aware Middleware for Resource Management in the Wireless Internet", *IEEE Transactions on Software Engineering*, vol. 29, no. 12, pp.1086–1099, 2003.

[12] P. C. David and T. Ledoux, "Towards a Framework for Self-Adaptive Component-Based Applications", *Proceedings of Distributed Applications and Interoperable Systems*, pages 1-14, 2003.

[13] J. Keeney and V. Cahill, "Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework", *Proceedings of IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, pp. 3-14, 2003.

[14] J. Sousaand D. Garlan, "Aura: An Architecture Framework for User Mobility in Ubiquitous Computing Environments", *Proceedings of 3rd IEEE Conference on Software Architecture*, pp. 29-43, 2002.

[15] Mobility and Adaptation Enabling Middleware (MADAM), Project Homepage: www.intermedia.uio.no/display/madam.

[16] B. Noble, "System Support for Mobile, Adaptive Applications", *IEEE Personal Communications*, pp. 44–49, 2000.

[17] B. D. Noble and M. Satyanarayanan, "Experience with Adaptive Mobile Applications in Odyssey", *Mobile Networks and Applications*, vol. 4, no. 4, pp. 245-254, 1999.

[18] P. C. David and T. Ledoux, "An Infrastructure for Adaptable Middleware", *Proceeding of the 2002 International Symposium on Distributed Objects and Applications* ,LNCS 2519, Springer-Verlag, pp.773-790, 2002.

[19] L. Capra, W. Emmerich, and C. Mascolo, "CARISMA: Context-Aware Reflective Middleware System for Mobile Applications", *IEEE Transactions on Software Engineering*, vol. 29, no. 10, pp. 929- 945, 2003.

[20] E. Rukzio, S. Siorpaes, O. Falke, and H. Hussmann, "Policy Based Adaptive Services for Mobile Commerce", *2nd Workshop on Mobile Commerce and, Munich, Germany*, pp. 183-192, July 19, 2005.

[21] A. K. Dey, G. Abowd and D. Salber "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications", *Human-Computer Interaction*, vol. 16 no. 2, pp. 97-166, 2001.

[22] A. Newberger and A. Dey, "Designer Support for Context Monitoring and Control", *Technical Report IRB-TR-03-017*, Intel Research, June, 2003.

[23] J. Robinson, I. Wakeman, and D. Chalmers, "Composing Software Services in the Pervasive Computing Environment: Languages or APIs?", *Pervasive and Mobile Computing*, vol. 4 no.4, pp. 481-505, August 2008.

[24] S. S. Yau, F. Karim, Y. Wang, B. Wang, and S. K. S. Gupta, "Reconfigurable Context-Sensitive Middleware for Pervasive Computing", *IEEE Pervasive Computing*, vol. 1, no. 3, pp. 33–40, 2002.

[25] K. Harihar and S. Kurkovsky, "Using Jini to enable pervasive computing environments", *Proceedings of the 43rd Annual Southeast Regional Conference*, pp. 188-193, 2005.

[26] Ponder2 Software and Documentation available at: http://www.ponder2.net/.

[27] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder Specification Language", *Workshop on Policies for Distributed Systems and Networks*, 2001.

[28] J. Fox and S. Clarke, "Exploring Approaches to Dynamic Adaptation", *Proceedings of the 3rd International DisccoTec Workshop on Middleware-Application interaction*, pp. 19-24, 2009.

[29] P. Bellavista, A. Corradi, and C. Stefanelli", "Mobile Agent Middleware for Mobile Computing*", IEEE Computer*, vol. 34, no. 3, March 2001.

[30] Sun Microsystems, " Remote Method Invocation", http://java.sun.com/products/jdk/rmi/index.html, 2001.

[31] Sun Microsystems, "Jini Architecture Specification", http://wwws.sun.com/software/jini/specs/jini1.2html/jini-spec.html.

[32] S. M. Sadjadi and P. K. McKinley, "ACT: An Adaptive CORBA Template to Support Unanticipated Adaptation", *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems*, Japan, March 2004.

[33] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects", *Theory and Practice of Object Systems*, vol. 3, no. 1, 1997.

[34] R. Vanegas, J. Zinky, J. Loyall, D. Karr, R. Schantz, and D. Bakken, "QuO's Runtime Support for Quality of Service in Distributed Objects", *Proceedings of Middleware 98, the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing*, September 1998.

[35] R. Friedman and E. Hadad, "Client Side Enhancements Using Portable Interceptors", *Proceedings of the 6th IEEE International Workshop on Object-oriented Real-time Dependable Systems*, January 2001.

[36] P. Tarr and H. Ossher, *Workshop on Advanced Separation of Concerns in Software Engineering at ICSE, May* 2001.

[37] R. Want, A. Hopper, V. Falcao, and J. Gibbons, "The Active Badge Location System", *ACM Transactions on Information Systems*, vol. 10 no.1, pp. 91-102, January 1992.

[38] B. Schilit, M. Theimer, and B. Welch, "Customizing Mobile Applications", *Proceedings of the USENIX Mobile and Location-Independent Computing Symposium*, pp. 129-138, Cambridge, Massachusetts, August 1993.

[39] B. Schilit and M. Theimer, "Disseminating Active Map Information to Mobile Hosts", *IEEE Network*, vol. 8, no. 5, pp. 22-32, 1994.

[40] M. Weiser, "Some Computer Science Issues in Ubiquitous Computing", *Communications of the ACM*, vol.36, pp.75-85, 1993.

[41] S. L. Keoh, K. Twidle, N. Pryce, A. E. Shaeffer-Filho, E. Lupu, N. Dulay, M. Sloman, S. Heeps, S. Strowes, J. Sventek, and E. Katsiri, "Policy-Based Management for Body-Sensor Networks", *Proceedings of 4th International Workshop on Wearable and Implantable Body Sensor Networks*, pp. 92 – 98, Aachen, Germany, March 2007.

[42] S. L. Keoh, N. Dulay, E. Lupu, K. Twidle, A. E. Schaeffer-Filho, M. Sloman, S. Heeps, S. Strowes, and J. Sventek, "Self-Managed Cell: A Middleware for Managing Body Sensor Networks", *Proceedings of International Conference on Mobile and Ubiquitous Systems: Networking and Services*, 2007.

[43] P. J. Brown, J. D. Bovey, and X. Chen, "Context-Aware Applications: From the Laboratory to the Marketplace", *IEEE Personal Communications*, vol.4 no.5 pp.58-64, October 1997.

[44] J. Flinn, M. Satyanarayanan, "Energy-Aware Adaptation for Mobile Applications", *Proceedings of the 17th ACM Symposium on Operating Systems and Principles*, December, 1999.

[45] O. Layaida, D. Hagimonte, "*Dynamic Adaptation in Distributed Multimedia Applications", INRIA, Technical Report*, August 2002.

[46] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras, " A Dynamic Reconfiguration Service for CORBA", *Proceedings of 4th International Conference on Configurable Distributed Systems*, pages 35–42, 1998.

[47] X. Chen and M. Simmons, "Extending RMI to Support Dynamic Reconfiguration of Distributed Systems", *Proceedings of 22nd InternationalConference on Distributed Computing Systems*, 2002.

[48] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management", *IEEE Transactions on Software Engineering*, vo. 16, no. 11, pp. 1293–1306, 1990.

[49] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based Runtime Software Evolution", *Proceedings of 20th International Conference on Software Engineering*, Kyoto, Japan, pp. 177-186, 1998.

[50] D. Ayed and Y. Berbers, "Dynamic Adaptation of CORBA Component-Based Applications", *Proceedings of the 2007 ACM symposium on Applied Computing*, pp. 580–585, 2007.

[51] P. Bellavista, A. Corradi, R. Montanari, and C. Stefanelli, "Dynamic Binding in Mobile Applications", *IEEE Internet Computing*, vol. 7, no. 2, pp. 34–42, March/April, 2003.

[52] A. Rascheand A. Polze, "Configuration and Dynamic Reconfiguration of Component-based Applications with Microsoft .NET*", Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2003.

[53] A. Carzaniga, G. P. Picco, and G. Vigna, "Designing Distributed Applications with Mobile Code Paradigms", *Proceedings of 19th Conference on Software Engineering*, pp. 22–32, 1997.

[54] A. Fuggetta, G.P. Picco, and G. Vigna, "Understanding Code Mobility", *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 342–361,1998.

[55] T. Elrad, R. E. Filman, and A. Bader, "Aspect-Oriented Programming: Introduction", *Communication of ACM*, vol. 44, no.10, pp. 29-32, October 2001.

[56] A. Nicoaraand G. Alonso, "Dynamic AOP with PROSE*", Proceedings of the International Workshop on Adaptive and Self-Managing Enterprise Applications,* 2005.

[57] P. Maes, "Concepts and Experiments in Computational Reflection*", Proceedings of OOPSLA Conference on Object-oriented Programming Systems and Applications*, pp.147–155, 1987.

[58] "Seamlessly Caching Stubs for Improved Performance", http://onjava.com/pub/a/onjava/2001/10/31/rmi.html.

[59] Object Management Group, "The Common Object Request Broker: Architecture and Specification", http://www.corba.org.

[60] The Component Object Model Specification, http://www.microsoft.com/com/default.mspx.

[61] D. Kulkarni and A. Tripathi, "Application-level Recovery Mechanisms for Context-Aware Pervasive Computing", *IEEE Symposium on Reliable Distributed Systems*, pp.13-22, 2008.

[62] G. Glass, "ObjectSpace Voyager Core Package Technical Overview", *ObjectSpace*, White Paper, 1999.

[63] O. Holder, I. Ben-Shaul, and H. Gazit. "System Support for Dynamic Layout of Distributed Applications", *Proceedings of the 19th International Conference on Distributed Computing Systems,* Austin, TX, USA, pp. 403-411, 1999.

[64] Z. Yu, I. Warren, and B. MacDonald, "Dynamic Reconfiguration for Robot Software", *Proceedings of the 2006 IEEE International Conference on Automation Science and Engineering*, Shanghai, China, pp. 292–297, 2006.

[65] IETF Policy Framework Working Group web page: www.ietf.org/html.charters/policy-charter.html/.

[66] R. Montanari, E. Lupu, and C. Stefanelli, "Policy-based Dynamic Reconfiguration of Mobile-Code Applications", *Computer, IEEE Computer Society*, vol. 37, no. 7, pp. 73-80, 2004.

[67] T. Zhang, "An Architecture for Building Customizable Context-Aware Applications by End-Users", *Proceedings of 2nd International Conference on Pervasive Computing,* 2004.

[68] J. Lobo, R. Bhatia,and S. Naqvi, "A Policy Description Language", *Proceedings of Innovative Applications of Artificial Intelligence*, pp. 291–298, 1999.

[69] J. Robinson, "The Exploration and Design of a Language and Middleware Architecture Dedicated to Service Composition in a Pervasive Computing Environment", Ph.D. Thesis, School of Informatics, University of Sussex, June 2006.

[70] N. Santos, P. Marques, and L. Silva, "A Framework for Smart Proxies and Interceptors in RMI", *Proceedings of 15th International Conference on Parallel and Distributed Computing Systems,* 2002.

[71] R. Kapitza, M. Kirstein, H. Schmidt, and F. J. Hauck, "FORMI: An RMI Extension for Adaptive Applications", *Proceedings of the 4th Workshop on Reflective and Adaptive Middleware*, New York, NY, USA, 2005.

[72] A. Stevenson and S. MacDonald, "Smart Proxies in Java RMI with Dynamic Aspect-oriented Programming", *Proceedings of the 2008 International Workshop on Java and Components for Parallelism, Distribution and Concurrency*, 2008.

[73] J. Groppe and W. Mueller, "Profile Management Technology for Smart Customizations in Private Home Applications", *Proceedings of International Workshop on Database and Expert Systems Applications*, pp. 226-230, 2005.

[74] C. Shin, A. K. Dey, and W. Woo, "Mixed-Initiative Conflict Resolution for Context-Aware Applications", *Proceedings of the International Conference on Ubiquitous Computing*, pp. 262-271, 2008.

[75] G.S. Thyagaraju, S. M. Joshi, U. P. Kulkarni, and S. K. NarasimhaMurthy, "Conflict Resolution in Multiuser Context-Aware Environments", *Proceedings of International Conference on Computational Intelligence for Modelling Control & Automation*, pp. 332-338, 2008.

[76] R. Grimm, J. Davis, B. Hendrickson, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad,G. Borriello, S. Gribble, and D. Wetherall,"Systems Directions for Pervasive Computing", *Proceedings of the 8th Workshop on Hot Topics in Operating Systems ,IEEE Computer Society*, pp. 147–151, 2001.

[77] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, S. Gribble, T. Anderson, B. Bershad, G.Borriello, D. Wetherall, "Programming for Pervasive Computing Environments", *University of Washington, Technical Report UW-CSE-01-06-01*, June 2001.

[78] R. Grimm, "One.World: Experiences with a Pervasive Computing Architecture", *IEEE Pervasive Computing*, vol. 3, no. 3, pp. 22-30, July-September 2004.

[79] T. Owen, I. Wakeman, B. Keller, J. Weeds, and D. Weir, "Managing the Policies of Non-Technical Users in a Dynamic World, *IEEE 6th International Workshop on Policies for Distributed Systems and Networks*, Stockholm, Sweden, 2005.

[80] K. Twidle, N. Dulay, E. Lupu, and M. Sloman, "Ponder2: A Policy System for Autonomous Pervasive Environments" *Proceedings of 5th International Conference Autonomic and Autonomous Systems*, pp. 330–335, April 2009.

[81] B. D. Noble, M. Price, and M. Satyanarayanan, "A Programming Interface for Application-Aware Adaptation in Mobile Computing", *Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, pp. 57-66, 1995.

[82] A. Mukhija and M. Glinz, "The CASA Approach to Autonomic Applications", *Proceedings of 5th IEEE Workshop on Applications and Services in Wireless Networks*, pp. 173-182, 2002.

[83] A. Mukhija and M. Glinz, "Runtime Adaptation of Applications through Dynamic Recomposition of Components" *Proceedings of 18th International Conference on Architecture of Computing Systems*, pp.124-138, 2005.

[84] R. Grimm and B. Bershad, "Future Directions: System Support for Pervasive Applications", *Proceedings of FuDiCo 2002: International Workshop on Future Directions in Distributed Computing*, Bertinoro, Italy, June 2002.

[85] M. Weiser, "The Computer for the Twenty-First Century", *Scientific American*, vol. 265, pp. 94–104, September 1991.

[86] M. Satyanarayanan, "Pervasive Computing: Vision and Challenges", *IEEE Personal Communication*, pp. 10-17, August 2001.

[87] The Open Group, "Architecture Description Markup Language (ADML)", http://www.opengroup.org/tech/architecture/adml/adml_home.htm.

[88] Object Management Group, "Unified Modelling Language Specification", http://www.omg.org/technology/documents.

[89] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, "Extensible Markup Language", http://www.w3.org/TR/1998/REC-xml-19980210, *World Wide Web Consortium*, March 1998.

[90] J. Magee, J. Kramer, "Dynamic Structure in Software Architectures", *Proceedings of the 4thACM SIGSOFT symposium on Foundations of software engineering*, pp. 3-14, 1996.

[91] D. Garlan, R.T. Monroe, D. Wile, "Acme: Architectural Description of Component-Based Systems", *Foundation of Component-based Systems*, pp. 47-67, 2000.

[92] P. K. McKinley, S. M. Sadjadi, E. P.Kasten, and B. H. C. Cheng, "Composing adaptive software", *IEEE Computer*, pp. 56-64, 2004.

[93] B. Win, F. Piessens, W. Joosen, and T. Verhanneman, "On the Importance of the Separation-of-Concerns Principle in Secure Software Engineering", *Workshop on the Application of Engineering Principles to System Security Design*, 2002.

[94] K. Twidle, N. Dulay, E. Lupu, and M. Sloman, " Ponder2 - A Policy Environment for Autonomous Pervasive Systems*", Proceedings of the 2008 IEEE Workshop on Policies for Distributed Systems and Networks*, pp. 245-246, 2008.

[95] J. Robinson, I. Wakeman, and T. Owen, "Scooby: Middleware for Service Composition in Pervasive Computing", *Proceedings of the 2nd Workshop on Middleware for Pervasive and Ad-hoc Computing*, pp. 161-166, 2004.

[96] N. Davies, J. Finney, A. Friday, and A. Scott, "Supporting Adaptive Video Applications in Mobile Environments", *IEEE Communications Magazine*, vol. 36, no. 6, pp.138-143, 1998.

[97] A. Fox, S. D. Gribble, Y. Chawathe, and E. A. Brewer, "Adapting to Network and Client Variation Using Infrastructural Proxies: Lessons and Perspectives", *IEEE Personal Communications*, 1998.

[98] A. Fox, S. D. Gribble, E. A. Brewer and E. Amir, "Adapting to Network and Client Variability via On-Demand Dynamic Distillation", *Proceedings of 7th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge*, MA, USA, 1996.

[99] Avis Event Router, http://avis.sourceforge.net/index.html.

[100] J. Rimmer, T. Owen, I. Wakeman B. Keller, J. Weeds, and D. Weir, "User Policies in Pervasive Computing Environments", *User Experience Design for Pervasive Computing, Pervasive Workshop*, 2005.

[101] B. Hardian, "Middleware Support for Transparency and User Control in Context-Aware Systems*", Proceedings of the 3rd International Middleware Doctoral Symposium*, Melbourne, Australia, 2006.

[102] E. Lupu, N. Dulay, M. Sloman, J. S. Sventek, S. Heeps, S. Strowes, K. P. Twidle, S. L. Keoh and A. E. S. Filho," AMUSE: Autonomic Management of Ubiquitous e-Health Systems" *Concurrency and Computation: Practice and Experience*, vol. 20, no. 3, pp. 277–295, 2008.

[103] E. Asmare and M. Sloman, "Self-management Framework for Unmanned Autonomous Vehicles" *Lecture Notes in Computer Science in Springer*, vol. 4543, pp. 164–167, 2007.